

## APP REPORT – PART 4

### TACHIYOMI

Link to the repository: <https://github.com/tachiyomior/tachiyomi>

Tachiyomi is a popular manga reader application that allows users to read manga on their mobile devices. Here are some scenarios that users might engage in while using the Tachiyomi app:

#### FIRST SCENARIO

##### Browsing and Reading Manga:

- User Action: Open the Tachiyomi app.
- System Response: Display the manga library with the mangas of the user.
- User Action: Browse different manga categories (e.g., Popular, Latest, Genres).
- System Response: Show a list of manga titles within the selected category.
- User Action: Select a specific manga.
- System Response: Display detailed information about the selected manga (cover, synopsis, chapters, etc.).
- User Action: Start reading a manga chapter.
- System Response: Present the manga pages with navigation options.

##### GPU Rendering Analysis:

Name	Total (µs)	%	Self (µs)	%	Children (µs)	%
RenderThread() ()	38,339,739	100.00	34,375,217	89.66	3,964,522	10.34
invokeDataAvailable() (android.graphicsHardwareRendererObserver)	3,961,677	10.33	0	0.00	3,961,677	10.33
get() (java.lang.ref.Reference)	2,600,211	6.78	2,600,211	6.78	0	0.00
notifyDataAvailable() (android.graphicsHardwareRendererObserver)	1,361,466	3.55	0	0.00	1,361,466	3.55
<init>() (android.graphicsHardwareRendererObserver\$\$ExternalSyntheticLambda0)	1,072,071	2.80	1,072,071	2.80	0	0.00
post() (android.os.Handler)	289,395	0.75	3,769	0.01	285,626	0.74
sendMessageDelayed() (android.os.Handler)	282,737	0.74	0	0.00	282,737	0.74
sendMessageAtTime() (android.view.ViewRootImpl\$ViewRootHandler)	282,737	0.74	0	0.00	282,737	0.74
sendMessageAtTime() (android.os.Handler)	282,737	0.74	276,814	0.72	5,923	0.02
getPostMessage() (android.os.Handler)	2,889	0.01	0	0.00	2,889	0.01
obtain() (android.os.Message)	2,889	0.01	2,889	0.01	0	0.00
callOnFinished() (android.graphics.animation.RenderNodeAnimator)	2,845	0.01	0	0.00	2,845	0.01
post() (android.os.Handler)	2,845	0.01	0	0.00	2,845	0.01
sendMessageDelayed() (android.os.Handler)	2,845	0.01	0	0.00	2,845	0.01
sendMessageAtTime() (android.os.Handler)	2,845	0.01	2,845	0.01	0	0.00

The provided GPU rendering analysis focus on the methods and their respective percentages of execution time in the RenderThread. Let's break down the information and discuss potential implications:

### **Methods Breakdown:**

The previous image about GPU rendering analysis focus on the methods and their respective percentages of execution time in the RenderThread. Let's break down the information and discuss potential implications:

#### **1. RenderThread() (100%):**

This represents the overall GPU rendering thread. The fact that it occupies 100% of the rendering time indicates that everything is executed within this thread.

#### **2. invokeDataAvailable() (10.33%):**

This method is called and contributes to about 10.33% of the rendering time. It could be related to data availability or updating.

#### **3. get() (6.78%):**

Within invokeDataAvailable(), the get() method is called and consumes 6.78% of the rendering time. This method typically involves retrieving data.

#### **4. notifyDataAvailable() (3.55%):**

This method, called within invokeDataAvailable(), is responsible for notifying that data is available. It takes up 3.55% of the rendering time.

#### **5. pos() (0.75%):**

This method, called within notifyDataAvailable(), is responsible for determining or setting a position. It occupies 0.75% of the rendering time.

#### **6. notifyOnFinished() (0.01%):**

This method, called within `invokeDataAvailable()`, is responsible for notifying that a process has finished. It takes up a minimal 0.01% of the rendering time.

## **Possible Issues:**

### **1. High Percentage in RenderThread:**

The fact that the entire GPU rendering time is spent in `RenderThread()` might indicate a single-threaded or synchronous rendering process. This could lead to performance bottlenecks, especially on devices with multiple cores.

### **2. InvokeDataAvailable Overhead:**

The `invokeDataAvailable()` method and its sub-methods (`get()`, `notifyDataAvailable()`, `pos()`, `notifyOnFinished()`) collectively contribute to a significant portion of the rendering time. This suggests that data availability and notification mechanisms are consuming a notable amount of GPU resources.

### **3. Data Retrieval Overhead:**

The `get()` method's relatively high percentage indicates that data retrieval might be a time-consuming operation. Consider optimizing data retrieval processes, perhaps through caching or asynchronous loading.

## **Possible Improvements:**

### **1. Multithreading:**

Introducing multithreading allows the application to execute tasks concurrently, utilizing multiple CPU cores. Therefore, it needs to identify tasks that can run independently and concurrently. For instance, separate UI rendering tasks from data retrieval operations. This can prevent one resource-intensive task from blocking the entire rendering process.

### **2. Async Operations:**

Making data-related operations asynchronous ensures that the rendering thread doesn't get blocked while waiting for potentially time-consuming operations to complete. Taking into account this, Tachiyomi could refactor the `get()` method to perform data retrieval asynchronously, possibly using Kotlin's coroutines or other asynchronous programming patterns. This ensures that the rendering thread remains responsive during data fetch operations.

### **3. Caching Strategies:**

Caching frequently used data helps reduce the need for repeated data retrieval, improving overall performance. In this way, Tachiyomi could implement an efficient caching mechanism for frequently accessed data. This can involve in-memory caching for quick access and, if applicable, persistent caching to survive app restarts. Evaluate different caching strategies based on data access patterns.

### **4. Background Processing:**

Moving non-rendering tasks, especially time-consuming ones like data retrieval, to background threads ensures a smooth user experience. Offload data retrieval tasks to background threads or services, allowing the rendering thread to focus on UI updates. Use Android's `AsyncTask`, Kotlin coroutines, or other background processing mechanisms to perform these tasks without impacting the main thread.

### **5. Profile and Test:**

Continuous profiling and testing are essential for identifying further opportunities for optimization and ensuring the application's stability and responsiveness. Also, utilize profiling tools to monitor the application's performance, identify bottlenecks, and track the impact of implemented changes. Conduct thorough testing under various scenarios, including different device specifications and network conditions, to ensure optimal performance across a range of use cases.

### **Overdrawing:**

Check Profiler Video E1 Overdrawing on Part 4 of the Web Version of the App-Report <https://danielfgmb.github.io/AppReport4/>

**Orange - Buffer Swap Stage:** Represents the time the CPU waits for the GPU to finish its work. A prolonged bar suggests that the GPU processing in the app is overly intensive.

**Red - Command Issue Stage:** Depicts the time Android's 2D renderer takes to send commands to OpenGL for drawing and redrawing display lists. The height of this bar is directly proportional to the sum of the time it takes to execute each display list (a greater number of display lists results in a longer red bar).

**Light blue - Sync and Load Stage:** Reflects the time needed to upload bitmap information to the GPU. A lengthy segment indicates that the app takes considerable time to load a significant amount of graphics.

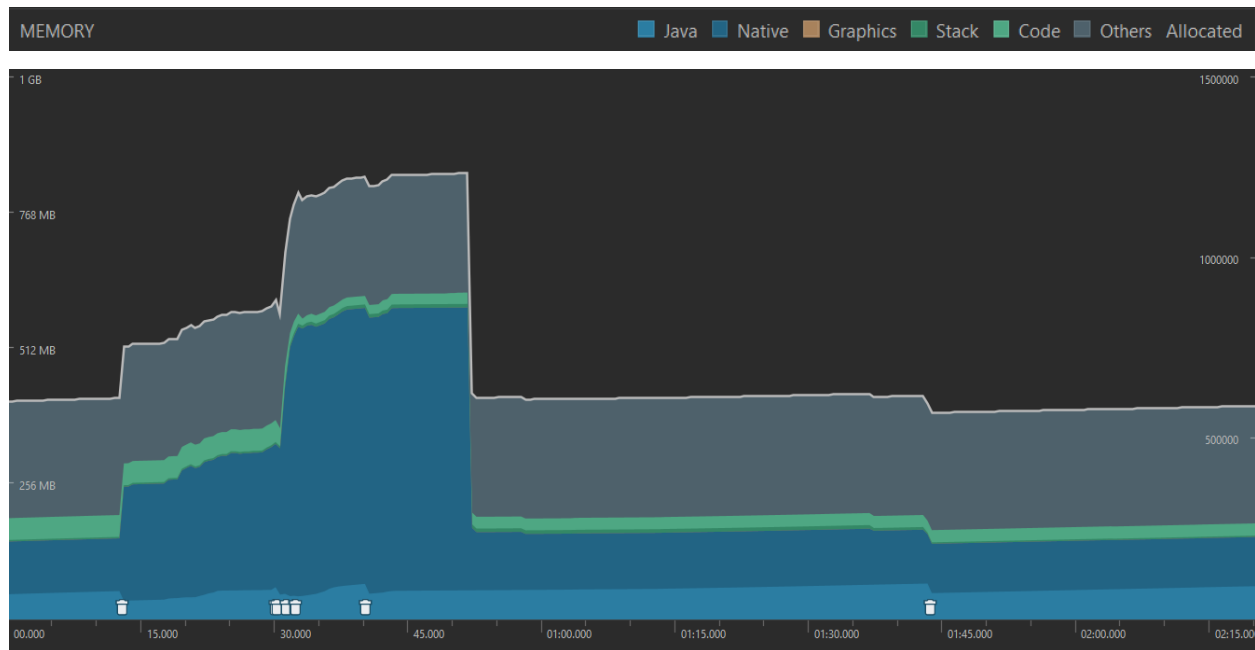
**Blue - Drawing Stage:** Illustrates the time used to create and update view display lists. If this part of the bar extends, it may suggest the drawing of many custom views or intensive processing in onDraw methods.

**Light green - Measure/Layout Stage:** Represents the amount of time spent in onLayout and onMeasure callbacks in the view hierarchy. A long segment indicates that the view hierarchy is taking a considerable time to process.

**Green - Input and Animation Handling:** Represents the time taken to evaluate all animators running for that frame and handle all input callbacks. A large segment could indicate that a custom animator or input callback is dedicating too much time to processing. View binding during scrolling, such as RecyclerView.Adapter.onBindViewHolder(), often occurs during this segment and is a more common source of delays in this stage.

**Cyan blue - Vertical Sync Delay/Miscellaneous Time:** Represents the time the app takes to execute operations between two consecutive frames. It could be an indicator of excessive processing on the UI thread, which could be offloaded to a different thread.

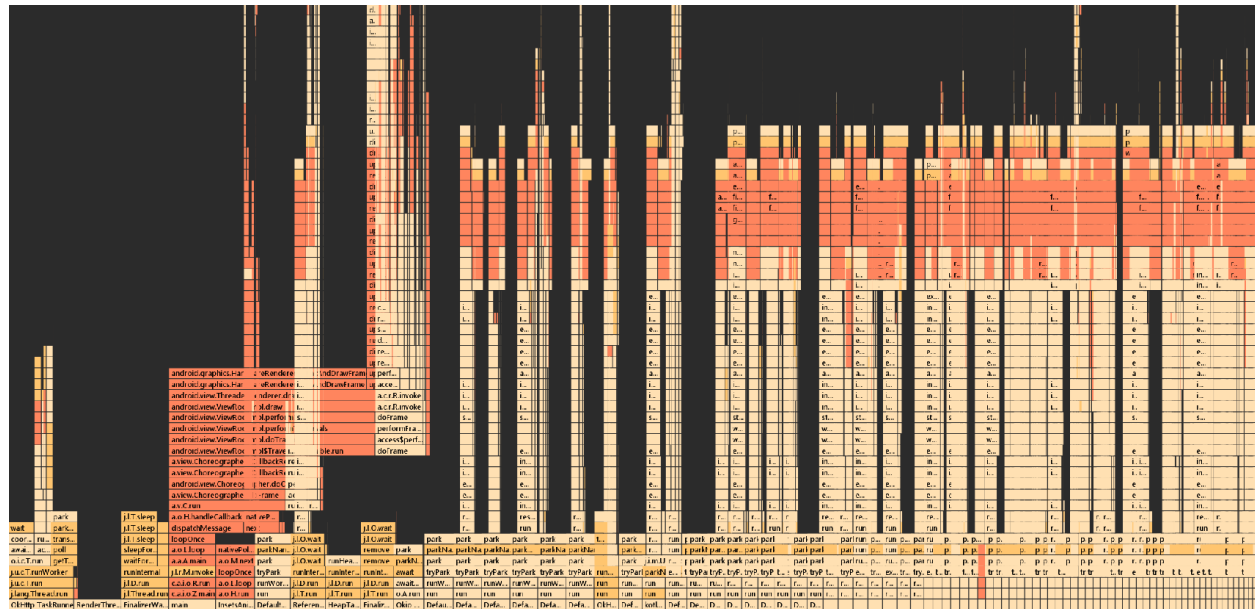
## Memory Management:



In the previous graph we could see the memory management of the Tachiyomi app when the user is navigating between screens and in particular following the path of the first scenario. We see a rise in the memory allocated for the Native. The reason of this is because the scenario 1 needs of data processing and storage and access to external libraries where the manga is. Also, at the moment of refreshing the UI switching between screens and the increasing number of threads in another reason for this increase in the memory making it go from less than 500mb to more than 750mb in barely seconds. After the data retrieval is finished then the memory starts to decrease.

The garbage collector is called 6 times along this process but curiously every time it's called the memory storage increases never decreases. Meaning that Tachiyomi every time is going to consume more memory calls the garbage collector. Also, the 3 almost simultaneously calls to the garbage collector give us an idea of a possible problem at the moment of using different sources of the app.

## Threading 110

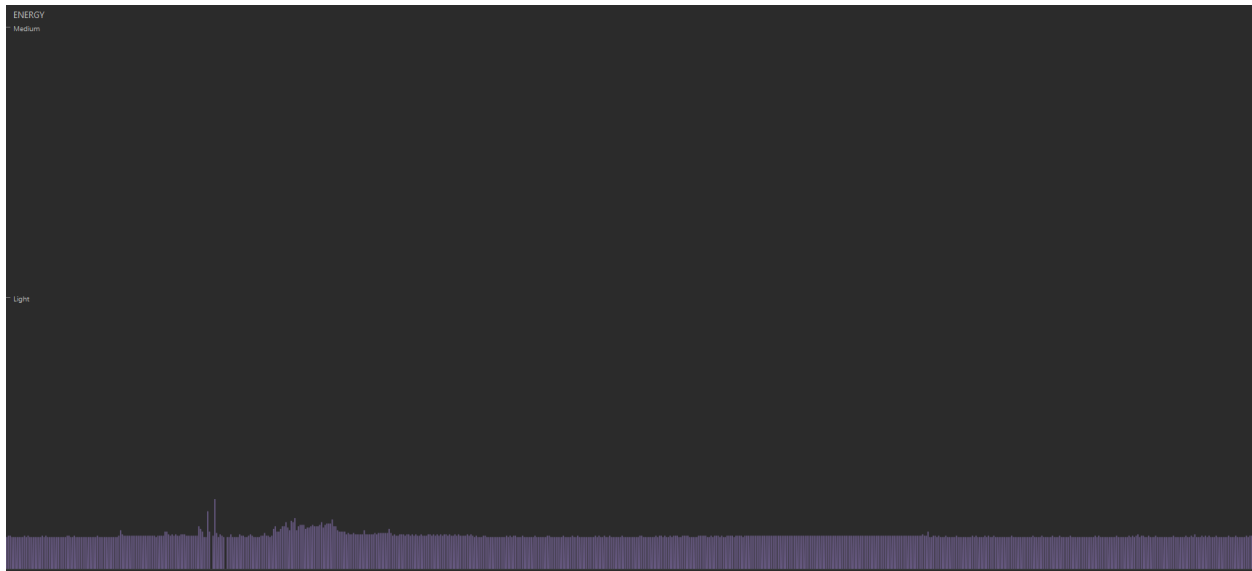


In this scenario, the deployment of 110 threads doesn't imply simultaneous execution or a direct impact on the main thread or UI. The complexity of this multithreading setup is driven by the crucial task of retrieving data from diverse external sources, especially when it involves multiple stages like gathering information and filtering down to the selected manga.

Within a multithreading architecture, threads collaborate to execute distinct tasks concurrently, including fetching and processing data, and updating the main thread or UI. The substantial thread count doesn't necessarily mean simultaneous execution; threads are instantiated, execute specific tasks, and then terminate upon completion. This dynamic thread management optimizes resource utilization and mitigates congestion.

The elevated thread count, illustrated by the 110 threads, results from the demanding process of extracting data from external sources. This involves multiple steps, each potentially executed on a separate thread for operational efficiency, such as initiating data requests and handling responses.

## **Energy Consumption:**



The energy consumption during screen transitions and events in the first scenario is consistently low. While there are occasional peaks, they are brief and momentary. On average, the energy consumption remains stable at a very low level.

In addition to low energy consumption during screen transitions and events, the Tachiyomi app demonstrates an efficient use of internet resources. The application minimizes unnecessary data fetches, optimizing its internet consumption. By strategically managing data retrieval processes, the app ensures that network requests are streamlined and focused, contributing to a responsive and energy-efficient user experience. This dual emphasis on both energy and internet efficiency reflects the app's commitment to providing users with a smooth and resource-conscious manga reading experience.

## **SECOND SCENARIO**

### **Managing Manga Library:**

- User Action: Open the Tachiyomi app.
- System Response: Display the manga library with the mangas of the user.
- User Action: Add a manga to the library.
- System Response: Include the manga in the user's library for easy access.



### GPU Rendering Analysis:

[illegible]

The provided GPU rendering analysis focus on the methods and their respective percentages of execution time in the RenderThread. Let's break down the information and discuss potential implications:

### Methods Breakdown:

### 1. RenderThread():

- The RenderThread is consuming 100% of the rendering time, indicating that the entire rendering process occurs within this thread.
- While it's expected for the rendering thread to be busy during UI updates, if this thread is consistently at 100%, it may indicate a lack of concurrency or potential bottlenecks.

## 2. invokeDataAvailable():

- This method is responsible for handling data availability.
- The method contributes 6.54% to rendering time. Depending on the nature of data availability events, this might be an acceptable or high percentage. If this is called frequently, consider optimizing data processing or introducing asynchronous operations.

### 3. notifyDataAvailable():

- This method notifies that data is available for processing.

- Consuming 4.51% of rendering time, it involves posting messages to the message queue. The high percentage may suggest that the application relies heavily on this mechanism. Optimizing the `post()` method might be beneficial.

#### **1. `post()`:**

- Posting messages to the message queue.
- Consuming 4.51% of rendering time. Optimizing this method, especially its nested methods, could lead to performance improvements.

##### **i. `sendMessageDelayed()`:**

- Sending delayed messages.
- Consuming 4.50% of rendering time. If excessive delayed messages are sent, it might impact responsiveness.

##### **ii. `getPostMessage()`:**

- Retrieving a post message.
- Consuming 0.01% of rendering time. While a small percentage, it contributes to the overall processing time.

##### **1. `obtain()`:**

- Obtaining a message object.
- Consuming 0.01% of rendering time. Although minor, optimizing object creation can have incremental benefits.

#### **4. `get()`:**

- Retrieving data.
- Consuming 2.03% of rendering time. Depending on the frequency and nature of data retrieval, consider optimizing this method, especially if it involves network or disk operations.

##### **1. `getReferent()`:**

- Getting the referent object.
- Consuming 2.02% of rendering time. Optimizing referent object retrieval can be beneficial, especially if it involves complex operations.

#### **5. `callOnFinished()`:**

- Calling when the rendering process is finished.
- Consuming only 0.01% of rendering time, indicating that the finishing process is relatively lightweight.

**1. post():**

- Posting a message when rendering is finished.
- Consuming 0.01% of rendering time. This part seems to have a minimal impact on performance.

**Possible Issues:**

**1. RenderThread Saturation:**

The RenderThread consistently operates at 100%, suggesting a potential lack of concurrency or presence of bottlenecks. That's why it needs concurrency to allow multiple tasks to be executed simultaneously and optimize data processing within the RenderThread to alleviate potential bottlenecks.

**2. High Percentage in notifyDataAvailable() and post():**

These methods contribute a significant percentage to rendering time, indicating heavy reliance on message posting mechanisms. Tachiyomi needs to optimize the mechanisms involved in message posting, such as reducing unnecessary notifications or improving the efficiency of handling data availability events. Also, evaluate the frequency of these events to ensure they align with application requirements.

**3. get() Method:**

If data retrieval is frequent or involves significant processing, it might impact rendering performance. The app needs to optimize the data retrieval process, considering asynchronous operations to prevent the RenderThread from being blocked during data fetch operations. Assessing the possibility of implementing caching strategies to reduce the need for repeated data retrieval.

## **Possible Improvements:**

### **1. Concurrency Introduction:**

- Introduce concurrency by identifying tasks that can be executed independently. This can potentially distribute the workload across multiple threads, preventing a single thread from saturating the RenderThread.
- Optimize processing tasks within the RenderThread to reduce the time each task takes, improving the overall efficiency of the rendering process.
- Investigate potential bottlenecks within the RenderThread and address them to enhance performance.

### **2. Message Posting Optimization:**

- Review the necessity and frequency of data availability notifications. Optimize the mechanism to ensure that notifications are only sent when essential.
- Improve the efficiency of the post() method to reduce the time spent in message posting. This may involve streamlining the process or identifying opportunities for optimization.
- Consider asynchronous handling of data availability events to minimize their impact on rendering time.

### **3. Data Retrieval Optimization**

- Optimize the data retrieval process within the get() method. Evaluate whether the method can be made asynchronous to avoid blocking the RenderThread during data fetch operations.
- Explore caching strategies to store frequently used data, reducing the need for repeated data retrieval.
- Assess the overall design and necessity of the get() method, considering potential improvements in data access patterns.

## **Overdrawing:**

Check Profiler Video E2 Overdrawing on Part 4 of the Web Version of the App-Report

<https://danielfgmb.github.io/AppReport4/>

[https://danielfgmb.github.io/AppReport4/vids/E2\\_Over.webm](https://danielfgmb.github.io/AppReport4/vids/E2_Over.webm)

**Orange - Buffer Swap Stage:** Denotes the duration the CPU patiently awaits the GPU to conclude its tasks. An elongated bar implies an overly demanding GPU processing within the application.

**Red - Command Issue Stage:** Illustrates the time taken by Android's 2D renderer to dispatch commands to OpenGL for the drawing and redrawing of display lists. The bar's height correlates directly with the cumulative time for each execution of a display list (a higher number of display lists results in an extended red bar).

**Light blue - Sync and Load Stage:** Reflects the time required to upload bitmap information to the GPU. A prolonged segment indicates that the app invests a significant amount of time in loading a considerable volume of graphics.

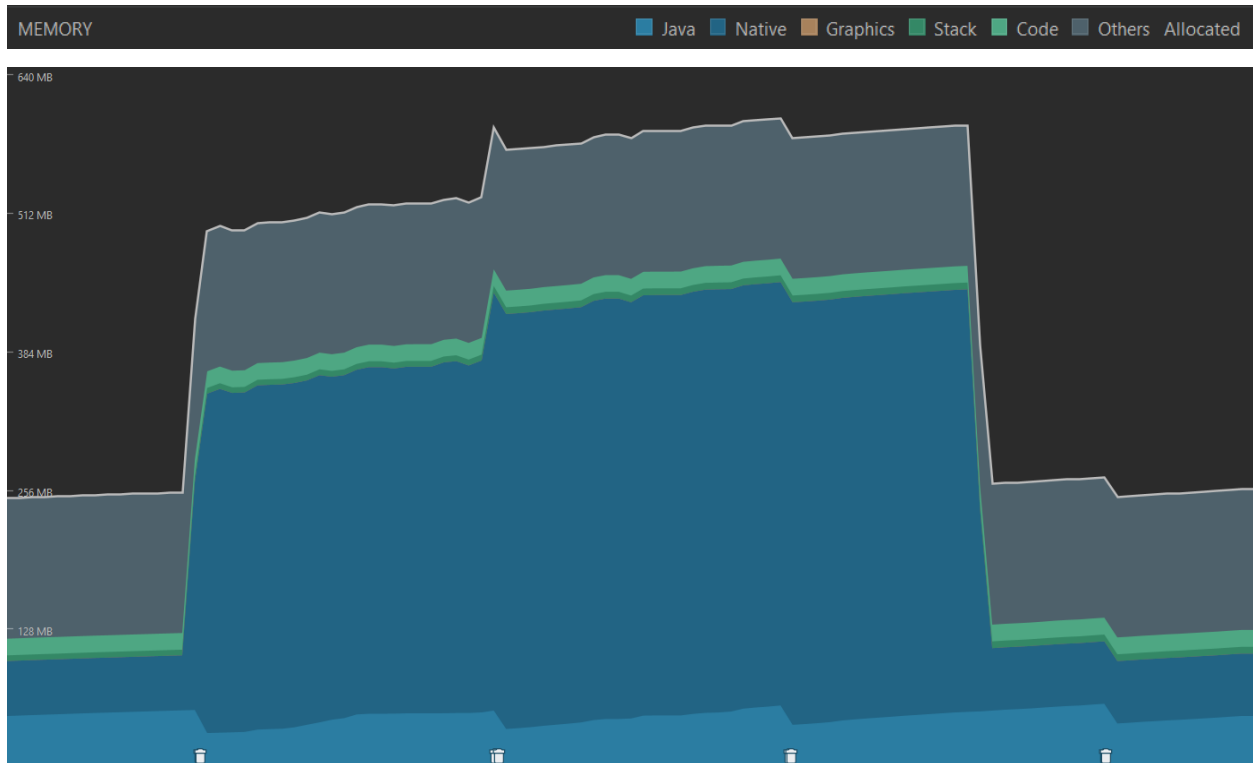
**Blue - Drawing Stage:** Highlights the time used for crafting and refreshing view display lists. If this segment of the bar extends, it may suggest the rendering of numerous custom views or intensive processing in onDraw methods.

**Light green - Measure/Layout Stage:** Represents the duration spent in onLayout and onMeasure callbacks within the view hierarchy. A lengthy segment signals that the view hierarchy is taking a substantial amount of time to process.

**Green - Input and Animation Handling:** Depicts the time invested in evaluating all animators running for a given frame and managing all input callbacks. A sizable segment could indicate that a custom animator or input callback is allocating an excessive amount of time to processing. Notably, view binding during scrolling, such as RecyclerView.Adapter.onBindViewHolder(), frequently occurs during this stage and serves as a common source of delays.

**Cyan blue - Vertical Sync Delay/Miscellaneous Time:** Signifies the time the app consumes to execute operations between two consecutive frames. It could suggest an excess of processing on the UI thread, which could potentially be offloaded to a different thread.

## Memory Management:

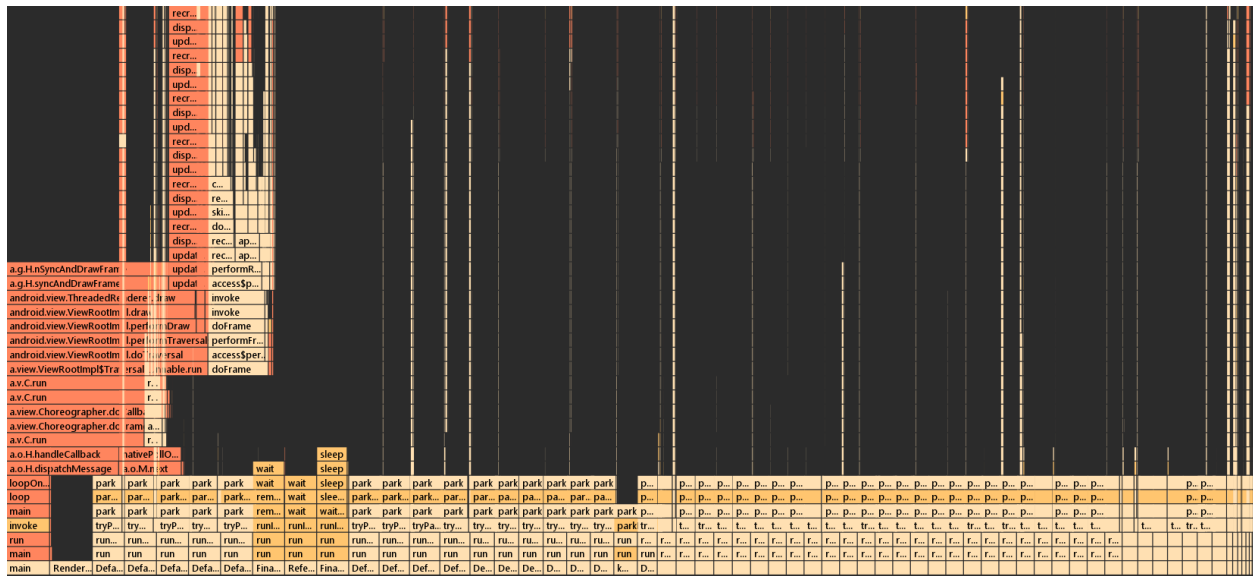


The surge in native memory might trigger more frequent garbage collector (GC) calls to free up memory and reclaim unused resources. While GC is essential for memory management, frequent calls can introduce performance overhead, causing intermittent pauses in the application.

Users might notice occasional stuttering or hiccups in the application's responsiveness, especially during garbage collection events. This can detract from the seamless and smooth user experience that is desirable in any application.

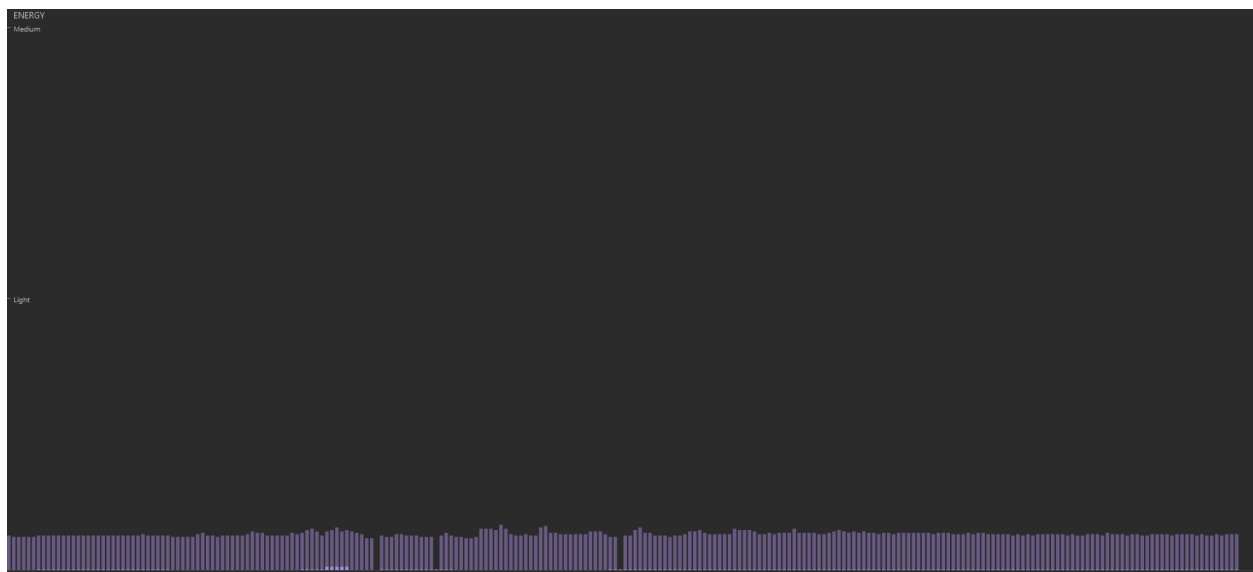
It is recommended to perform a thorough code review to identify areas where native memory is allocated and deallocated. Also, consider tuning garbage collector settings to strike a balance between reclaiming memory and minimizing the impact on application responsiveness.

## Threading 87



In this graph we can observe the methods that used more time in processing. The methods in this case are the choreographer that coordinates the timing of animations, input and drawing, so is understandable that uses more time than the other methods. So although, the app creates a significant number of threads (87) in this specific task, the methods related to the activity are handled efficiently because none of them are taking too much time in CPU processing time.

### Energy Consumption:



In an environment devoid of peaks, the energy consumption throughout the entire scenario remains consistently constant. This sustained stability reflects Tachiyomi's dedication to maintaining a steady and low level of energy usage, contributing to an uninterrupted and energy-efficient manga reading experience. The absence of notable fluctuations in energy consumption underlines the app's commitment to providing users with a seamlessly optimized performance across different scenarios.

## THIRD SCENARIO

### Customizing Reading Experience:

- User Action: Open the Tachiyomi app.
- System Response: Display the manga library with the mangas of the user.
- User Action: Adjust reading settings (e.g., brightness, orientation, reading direction).
- System Response: Apply the selected settings to enhance the reading experience.
- User Action: Change reading modes (e.g., single page, double page).
- System Response: Modify the display layout according to the selected mode.

### GPU Rendering Analysis:

Name	Total (µs)	%	Self (µs)	%	Children (µs)	%
RenderThread() ()	70,797,890	100.00	63,931,856	90.30	6,866,034	9.70
onFrameCommit() (android.view.ThreadedRenderer\$ExternalSyntheticLambda0)	2,912	0.00	0	0.00	2,912	0.00
lambda\$onFrameDraw\$00() (android.view.ThreadedRenderer\$1)	2,912	0.00	0	0.00	2,912	0.00
get() (java.util.ArrayList)	2,912	0.00	2,912	0.00	0	0.00
invokeDataAvailable() (android.graphics.HardwareRenderer\$Observer)	6,861,536	9.68	1,891,455	2.67	4,969,081	7.01
notifyDataAvailable() (android.graphics.HardwareRenderer\$Observer)	4,961,322	6.99	21,533	0.03	4,939,789	6.96
post() (android.os.Handler)	4,927,913	6.96	14,161	0.02	4,913,752	6.94
sendMessageDelayed() (android.os.Handler)	4,913,752	6.94	0	0.00	4,913,752	6.94
sendMessageAtTime() (android.view.ViewRootImpl\$ViewRootHandler)	4,913,752	6.94	1,936,297	2.73	2,977,455	4.21
sendMessageAtTime() (android.os.Handler)	2,977,455	4.21	0	0.00	2,977,455	4.21
enqueueMessage() (android.os.Handler)	2,977,455	4.21	0	0.00	2,977,455	4.21
get() (android.os.ThreadLocalWorkSource)	2,083,751	2.94	3,762	0.01	2,079,989	2.94
get() (java.lang.ThreadLocal)	2,079,989	2.94	7,622	0.01	2,072,367	2.93
currentThread() (java.lang.Thread)	167,031	0.24	167,031	0.24	0	0.00
isNextSyncEntry() (java.lang.ThreadLocal\$ThreadLocalMap)	1,905,336	2.69	1,893,302	2.67	12,034	0.02
getEntry() (java.lang.ThreadLocal\$ThreadLocalMap)	12,034	0.02	0	0.00	12,034	0.02
referTo() (java.lang.ref.Reference)	12,034	0.02	7,358	0.01	4,676	0.01
referTo() (java.lang.ref.Reference)	4,676	0.01	4,676	0.01	0	0.00
enqueueMessage() (android.os.MessageQueue)	893,704	1.26	893,704	1.26	0	0.00
<init>() (android.graphics.HardwareRenderer\$ExternalSyntheticLambda0)	1,876	0.00	1,876	0.00	0	0.00
get() (java.lang.ref.Reference)	8,349	0.01	0	0.00	8,349	0.01
getReferent() (java.lang.ref.Reference)	8,349	0.01	8,349	0.01	0	0.00
callOnFinished() (android.graphics.animation.RenderNodeAnimator)	11,596	0.02	0	0.00	11,596	0.02
requireNonNull() (java.util.Objects)	3,857	0.01	3,857	0.01	0	0.00
post() (android.os.Handler)	3,025	0.00	0	0.00	3,025	0.00
sendMessageDelayed() (android.os.Handler)	3,025	0.00	0	0.00	3,025	0.00
sendMessageAtTime() (android.os.Handler)	3,025	0.00	3,025	0.00	0	0.00
<init>() (android.graphics.animation.RenderNodeAnimator\$ExternalSyntheticLambda0)	4,714	0.01	4,714	0.01	0	0.00



The provided GPU rendering analysis focus on the methods and their respective percentages of execution time in the RenderThread. Let's break down the information and discuss potential implications:

### **Methods Breakdown:**

#### **1. RenderThread() (100%):**

Represents the overall GPU rendering thread. It occupies 100% of the rendering time, indicating that all rendering operations are performed within this thread.

#### **2. onFrameCommit() (0.001%):**

A method called within the RenderThread, contributing a negligible 0.001% to the rendering time. It likely involves actions related to committing frames.

#### **3. invokeDataAvailable() (9.68%):**

This method, called within the RenderThread, contributes to about 9.68% of the rendering time. It might be associated with data availability or updating processes.

#### **4. notifyDataAvailable() (6.99%):**

A sub-method called within invokeDataAvailable(). It is responsible for notifying that data is available and occupies 6.99% of the rendering time.

#### **5. post() (6.96%):**

A sub-method of notifyDataAvailable(), the post() method handles message posting. It contributes to 6.96% of the rendering time.

#### **6. sendMessageDelayed() (6.94%):**

A more specific sub-method within post(). It involves sending delayed messages and occupies 6.94% of the rendering time.

#### **7. get() (0.01%):**

A method called within `invokeDataAvailable()`, responsible for data retrieval. It consumes a minimal 0.01% of the rendering time.

**8. `getReferent()` (0.01%):**

A sub-method of `get()`, involved in obtaining a reference. It occupies 0.01% of the rendering time.

**9. `callOnFinished()` (0.02%):**

A method called within `invokeDataAvailable()`, responsible for indicating that a process has finished. It takes up a minimal 0.02% of the rendering time.

**10. `requireNonNull()` (0.01%):**

A method called within `callOnFinished()`, involving the requirement of a non-null value. It occupies 0.01% of the rendering time.

**11. `post()` (0.01%):**

Another instance of the `post()` method, called within `callOnFinished()`, contributing 0.01% to the rendering time.

**Possible Strengths:**

**1. Efficiency in Frame Commitment:**

The `onFrameCommit()` method's minimal contribution suggests efficiency in handling frame commitments, which is crucial for smooth rendering.

**Possible Issues:**

**1. High Percentage in `NotifyDataAvailable()` and `Post()`:**

These methods, especially `notifyDataAvailable()` and its sub-methods, contribute significantly to rendering time. This may indicate potential inefficiencies in data availability notification and message posting mechanisms.

**2. Data Retrieval Overhead:**

The `get()` method, although minimal, involves data retrieval and might impact rendering performance.

### **3. Redundant Post() Calls:**

Multiple instances of the `post()` method within different contexts might indicate redundancy.

### **4. Nested Method Calls in `invokeDataAvailable()`:**

Multiple levels of method calls within `invokeDataAvailable()` may contribute to a significant portion of rendering time.

## **Possible Improvements:**

### **1. High Percentage in `NotifyDataAvailable()` and `Post()`:**

Optimize the message posting process, considering whether the frequency of data availability events justifies their impact on rendering time. Implement asynchronous handling if applicable.

### **2. Data Retrieval Overhead:**

Assess the necessity and frequency of data retrieval within the rendering thread. Optimize data retrieval processes, and consider asynchronous operations to prevent blocking the `RenderThread`.

### **3. Redundant Post() Calls:**

Review the necessity of each `post()` call, eliminating redundancy, and optimizing the message posting mechanism.

### **4. Nested Method Calls in `invokeDataAvailable()`:**

Evaluate the hierarchy of method calls within `invokeDataAvailable()` and optimize the structure to reduce overhead.

## **Overdrawing:**

Check Profiler Video E3 Overdrawing on Part 4 of the Web Version of the App-Report

<https://danielfgmb.github.io/AppReport4/>

[https://danielfgmb.github.io/AppReport4/vids/E3\\_Over.webm](https://danielfgmb.github.io/AppReport4/vids/E3_Over.webm)

**range - Buffer Swap Stage:** Portrays the time the CPU patiently waits for the GPU to complete its tasks. An extended bar suggests an overly demanding GPU processing within the application.

**Red - Command Issue Stage:** Depicts the time taken by Android's 2D renderer to dispatch commands to OpenGL for drawing and redrawing display lists. The bar's height correlates directly with the cumulative time for each execution of a display list (a higher number of display lists results in an extended red bar).

**Light blue - Sync and Load Stage:** Reflects the time needed to upload bitmap information to the GPU. A prolonged segment indicates that the app takes considerable time to load a significant amount of graphics.

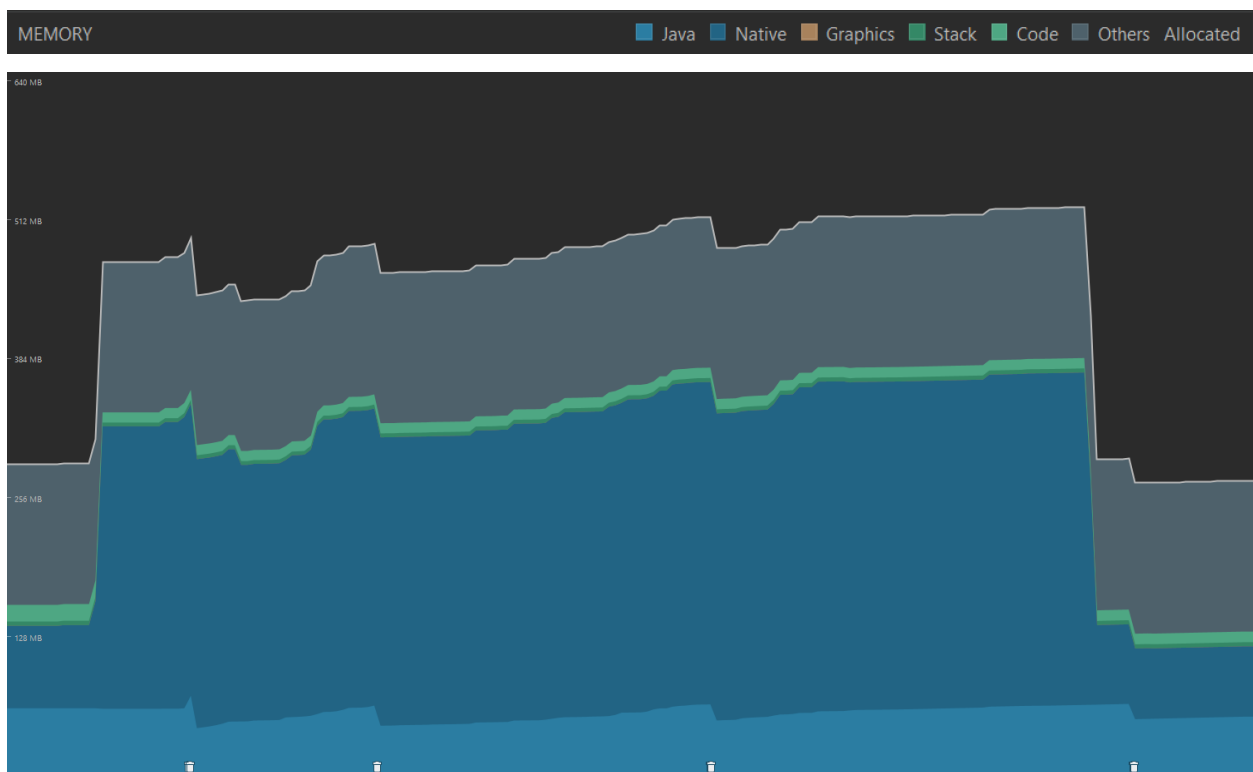
**Blue - Drawing Stage:** Illustrates the time used to create and update view display lists. If this part of the bar extends, it may suggest the drawing of many custom views or intensive processing in onDraw methods.

**Light green - Measure/Layout Stage:** Represents the amount of time spent in onLayout and onMeasure callbacks in the view hierarchy. A long segment indicates that the view hierarchy is taking a considerable time to process.

**Green - Input and Animation Handling:** Represents the time taken to evaluate all animators running for that frame and handle all input callbacks. A large segment could indicate that a custom animator or input callback is dedicating too much time to processing. View binding during scrolling, such as RecyclerView.Adapter.onBindViewHolder(), often occurs during this segment and is a more common source of delays in this stage.

**Cyan blue - Vertical Sync Delay/Miscellaneous Time:** Signifies the time the app takes to execute operations between two consecutive frames. It could be an indicator of excessive processing on the UI thread, which could be offloaded to a different thread.

### Memory Management:

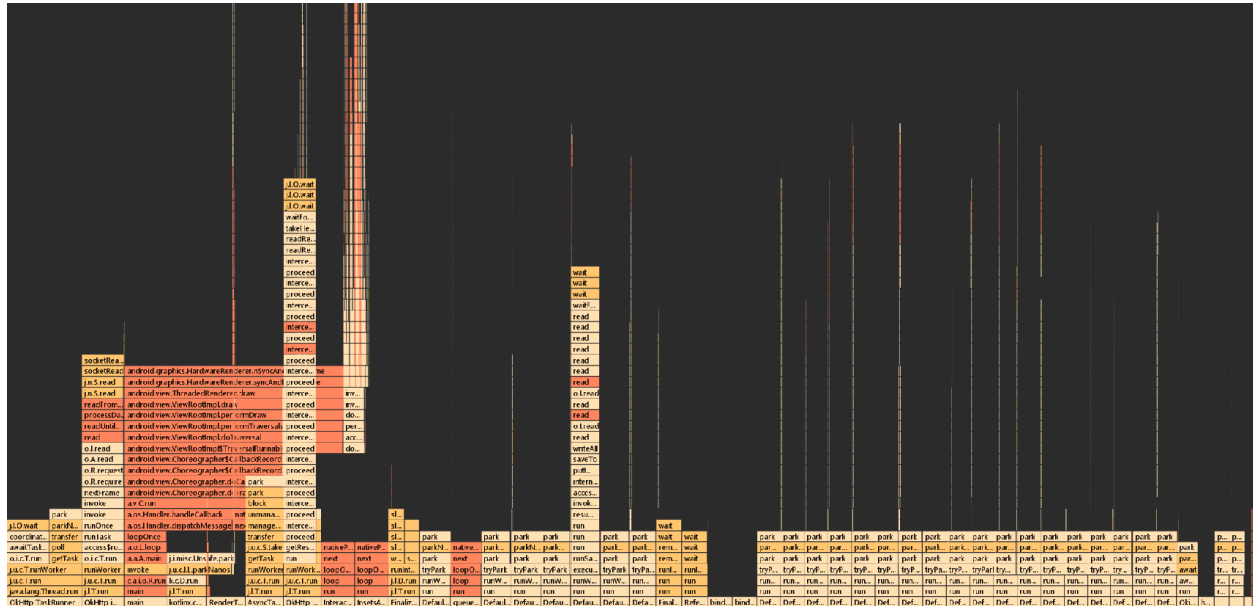


The escalated memory usage may push the application closer to memory limits, increasing the likelihood of out-of-memory (OOM) errors. This can result in app crashes, leading to a poor user experience and potential frustration for users.

High native memory usage often correlates with increased CPU usage, contributing to higher energy consumption. This can lead to accelerated battery drain, negatively affecting the device's battery life.

Garbage collection introduces CPU overhead as it involves scanning and reclaiming memory. With frequent invocations, the cumulative impact on CPU usage can be substantial, potentially affecting the overall responsiveness of the application.

## Threading 60



In the profiler's findings, some methods stand out for taking more time than in the previous scenario. One of these is the choreographer, which handles tasks like coordinating animations and input. It makes sense that this would take longer, considering its role. Surprisingly, despite using fewer threads (60) for these tasks, the main functions still work efficiently. None of them seem to put too much strain on the CPU, showing smart resource management in the app.

## Energy Consumption:



The energy usage during transitions and events in the first scenario consistently remains low. There is no kind of significant peak, contributing to a smooth and uninterrupted user interface. On average, the energy consumption level stays stable and minimal, ensuring an overall seamless user experience.

Additionally, Tachiyomi adeptly manages internet resources without experiencing any kind of peak. The app minimizes unnecessary data retrievals, optimizing overall internet consumption. Smart data retrieval processes ensure streamlined network requests, enhancing efficiency in both energy and internet usage. This commitment reflects the app's dedication to providing users with an optimized manga reading experience. Importantly, no significant peak is observed throughout the entire scenario.

## **FORTH SCENARIO**

### **Discovering New Manga:**

- User Action: Open the Tachiyomi app.
- System Response: Display the manga library with the mangas of the user.
- User Action: Search for a specific manga title.
- System Response: Display search results matching the entered query.
- User Action: Filter for the chapter that has not been read.
- System Response Shows the chapters of the manga that matches the query.

**GPU Rendering Analysis:**

Name	Total (µs)	%	Self (µs)	%	Children (µs)	%
RenderThread() ()	38,251,484	100.00	36,241,661	94.75	2,009,833	5.25
invokeDataAvailable() (android.graphicsHardwareRendererObserver)	2,009,833	5.25	499,892	1.20	1,549,941	4.05
get() (java.lang.ref.Reference)	1,349,045	3.53	15,459	0.04	1,333,586	3.49
getReferent() (java.lang.ref.Reference)	1,333,586	3.49	1,333,586	3.49	0	0.00
notifyDataAvailable() (android.graphicsHardwareRendererObserver)	200,896	0.53	4,573	0.01	196,323	0.51
post() (android.os.Handler)	196,323	0.51	0	0.00	196,323	0.51
sendMessageDelayed() (android.os.Handler)	193,076	0.50	0	0.00	193,076	0.50
sendMessageAtTime() (android.view.ViewRootImpl\$ViewRootHandler)	193,076	0.50	21,028	0.05	172,048	0.45
sendMessageAtTime() (android.os.Handler)	172,048	0.45	0	0.00	172,048	0.45
enqueueMessage() (android.os.Handler)	172,048	0.45	0	0.00	172,048	0.45
getUId() (android.os.ThreadLocalWorkSource)	172,048	0.45	0	0.00	172,048	0.45
get() (java.lang.ThreadLocal)	172,048	0.45	1,684	0.00	170,364	0.45
\$setValueEntity() (java.lang.ThreadLocal\$ThreadLocalMap)	165,341	0.43	160,994	0.42	4,347	0.01
getEntry() (java.lang.ThreadLocal\$ThreadLocalMap)	4,347	0.01	0	0.00	4,347	0.01
referTo() (java.lang.ref.Reference)	4,347	0.01	1,796	0.00	2,551	0.01
referTo() (java.lang.ref.Reference)	2,551	0.01	2,551	0.01	0	0.00
currentThread() (java.lang.Thread)	5,023	0.01	5,023	0.01	0	0.00
getPostMessage() (android.os.Handler)	3,247	0.01	0	0.00	3,247	0.01
obtain() (android.os.Message)	3,247	0.01	3,247	0.01	0	0.00

The provided GPU rendering analysis focus on the methods and their respective percentages of execution time in the RenderThread. Let's break down the information and discuss potential implications:

**Methods Breakdown:**

**1. RenderThread() (100%):**

Represents the overall GPU rendering thread. It occupies 100% of the rendering time, indicating that all rendering operations are performed within this thread.

**2. invokeDataAvailable() (5.25%):**

A method called within the RenderThread, contributing to about 5.25% of the rendering time. It might be associated with data availability or updating processes.

**3. get() (3.53%):**

A sub-method called within invokeDataAvailable(), responsible for data retrieval. It consumes 3.53% of the rendering time.

**4. getReferent() (3.49%):**

Description: A more specific sub-method within get(), involved in obtaining a reference. It occupies 3.49% of the rendering time.



**5. notifyDataAvailable() (0.53%):**

A method called within invokeDataAvailable(). It is responsible for notifying that data is available and occupies 0.53% of the rendering time.

**6. post() (0.51%):**

A sub-method of notifyDataAvailable(), the post() method handles message posting. It contributes to 0.51% of the rendering time.

**7. sendMessageDelayed() (0.50%):**

A more specific sub-method within post(). It involves sending delayed messages and occupies 0.50% of the rendering time.

**8. getPostMessage() (0.01%):**

A sub-method called within post(), responsible for obtaining posted messages. It occupies 0.01% of the rendering time.

**9. obtain() (0.01%):**

A method called within getPostMessage(), involving the obtaining of a message. It occupies 0.01% of the rendering time.

**Possible Strengths:**

**1. Efficient Data Retrieval:**

The get() method, including getReferent(), consumes a significant portion of rendering time, indicating efficient data retrieval processes.

**Possible Issues:**

**1. Nested Method Calls in invokeDataAvailable():**

Multiple levels of method calls within invokeDataAvailable() may contribute to a significant portion of rendering time.

## **2. Relatively High Percentage in NotifyDataAvailable() and Post():**

Although relatively low, the notifyDataAvailable() and post() methods contribute to rendering time.

## **3. Specific Sub-Methods with Low Contribution:**

Methods like getPostMessage() and obtain() have a minimal impact on rendering time but may add complexity.

## **Possible Improvements:**

### **1. Nested Method Calls in invokeDataAvailable():**

Evaluate the hierarchy of method calls within invokeDataAvailable() and optimize the structure to reduce overhead.

### **2. Relatively High Percentage in NotifyDataAvailable() and Post():**

Review the necessity of each call, optimizing the message posting process, and considering asynchronous handling if applicable.

### **3. Specific Sub-Methods with Low Contribution:**

Assess the necessity of such specific sub-methods and consider simplifying or optimizing the message handling process.

## **Overdrawing:**

Check Profiler Video E4 Overdrawing on Part 4 of the Web Version of the App-Report

<https://danielfgmb.github.io/AppReport4/>

[https://danielfgmb.github.io/AppReport4/vids/E3\\_Over.webm](https://danielfgmb.github.io/AppReport4/vids/E3_Over.webm)

**range - Buffer Swap Stage:** Indicates the time the CPU awaits the completion of GPU tasks. An extended bar suggests intensive GPU processing within the application.

**Red - Command Issue Stage:** Illustrates the time Android's 2D renderer takes to dispatch commands to OpenGL for drawing and redrawing display lists. The bar's height correlates directly with the cumulative time for each execution of a display list (a higher number of display lists results in an extended red bar).

**Light blue - Sync and Load Stage:** Reflects the time required to upload bitmap information to the GPU. A prolonged segment indicates that the app invests a significant amount of time in loading a considerable volume of graphics.

**Blue - Drawing Stage:** Highlights the time used for crafting and refreshing view display lists. If this segment of the bar extends, it may suggest the rendering of numerous custom views or intensive processing in onDraw methods.

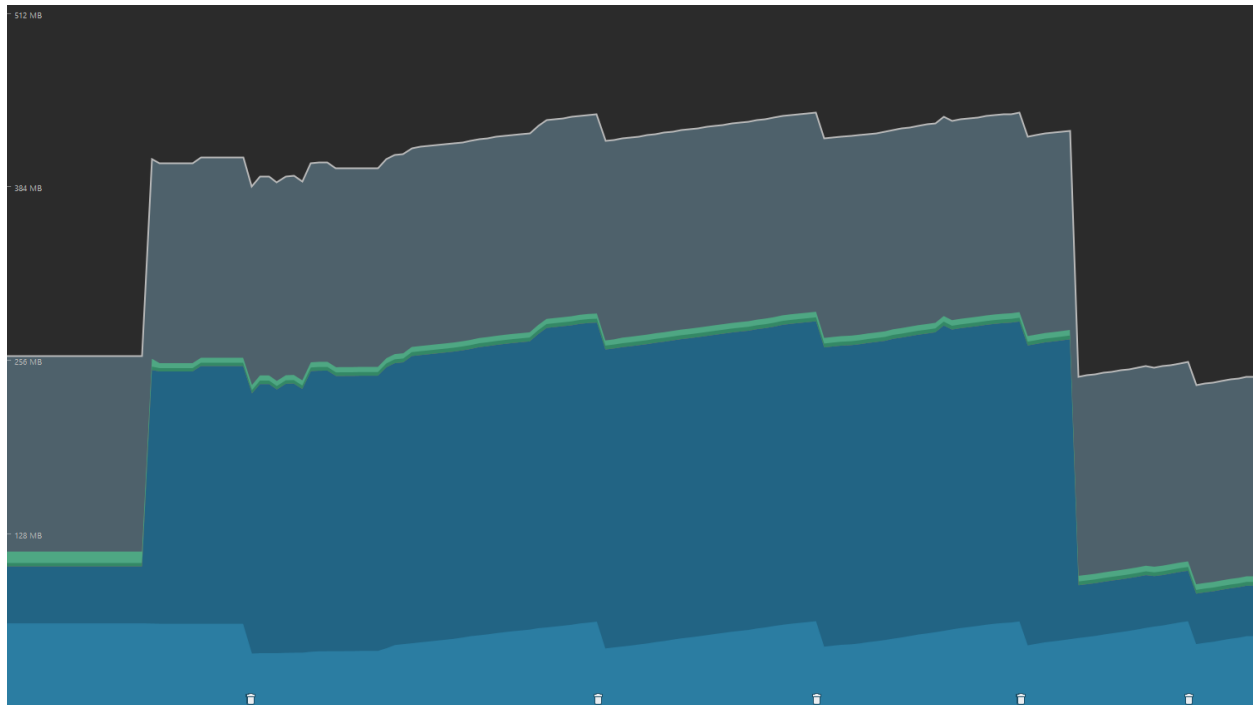
**Light green - Measure/Layout Stage:** Represents the duration spent in onLayout and onMeasure callbacks within the view hierarchy. A lengthy segment signals that the view hierarchy is taking a substantial amount of time to process.

**Green - Input and Animation Handling:** Depicts the time invested in evaluating all animators running for a given frame and managing all input callbacks. A sizable segment could indicate that a custom animator or input callback is allocating an excessive amount of time to processing. Notably, view binding during scrolling, such as RecyclerView.Adapter.onBindViewHolder(), frequently occurs during this stage and serves as a common source of delays.

**Cyan blue - Vertical Sync Delay/Miscellaneous Time:** Signifies the time the app consumes to execute operations between two consecutive frames. It could suggest an excess of processing on the UI thread, which could potentially be offloaded to a different thread.

### **Memory Management:**





In the given scenario, a substantial increase in native memory usage has been observed. Such a surge in native memory consumption can have several potential consequences for the application's performance and overall user experience.

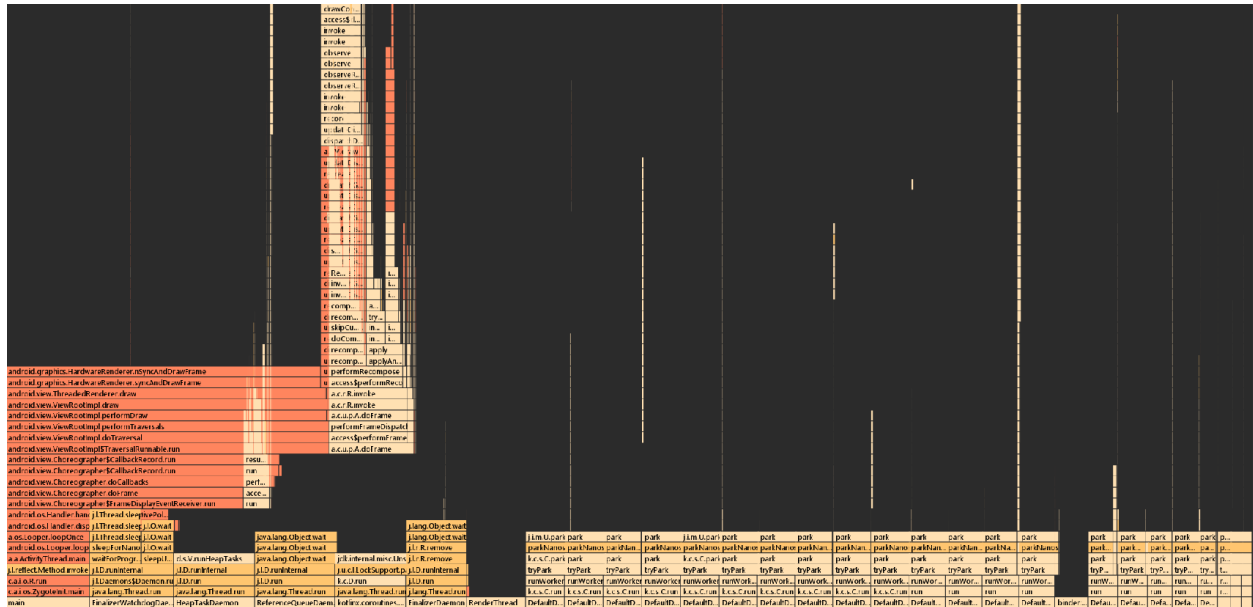
With a significant rise in native memory, the overall performance of the application may suffer. Increased memory usage can lead to slower response times, longer loading durations, and a generally sluggish user interface.

The excessive use of native memory can lead to resource contention, impacting the ability of the application to efficiently utilize system resources. This contention may extend to other components or processes, causing a ripple effect on the device's overall performance.

In addition to the increased native memory usage, the fact that the garbage collector was called five times further emphasizes the memory management challenges faced by the application. Frequent garbage collector invocations can result in the following:

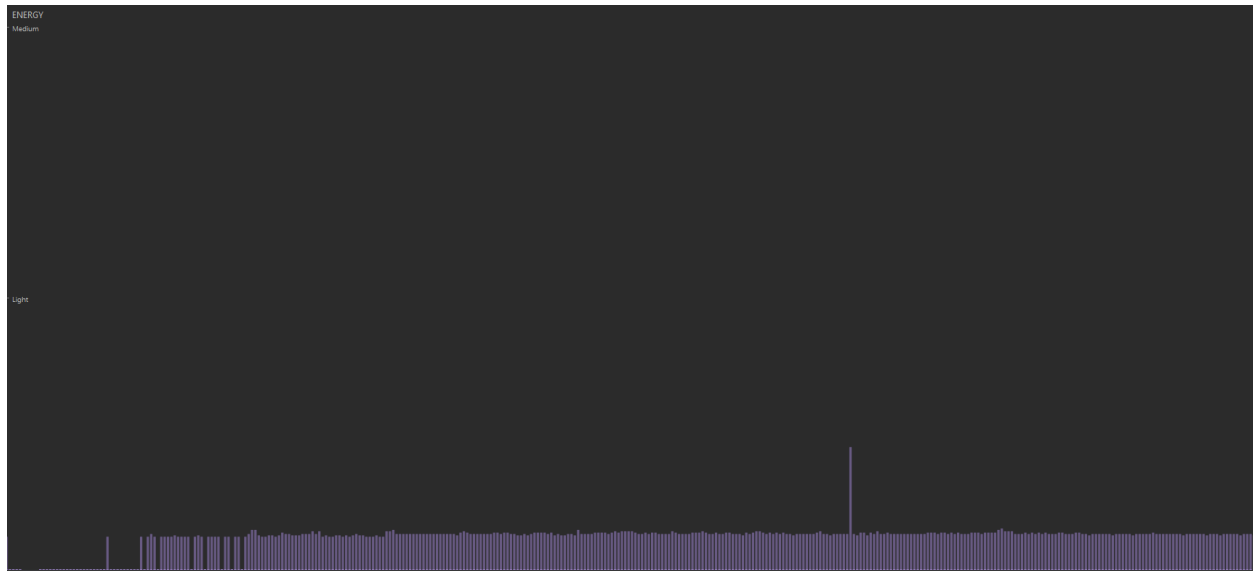
The garbage collector temporarily halts the application's execution to identify and collect unused memory. As a consequence, users may experience intermittent pauses or delays during these collection cycles.

## Threading 48



Just like before, some tasks take longer according to the profiler. One of them is the choreographer, which handles animations and input. Even though it uses fewer threads (48) for these jobs, the main functions still work well. None of these tasks seem to stress out the CPU, showing that the app manages resources similarly in different situations.

### Energy Consumption:



The energy consumption during screen transitions and events in the first scenario remains consistently low. While there are occasional peaks, they are brief and momentary. On average, the energy consumption level stays stable and very low. This energy efficiency contributes to an overall smooth and responsive user experience.

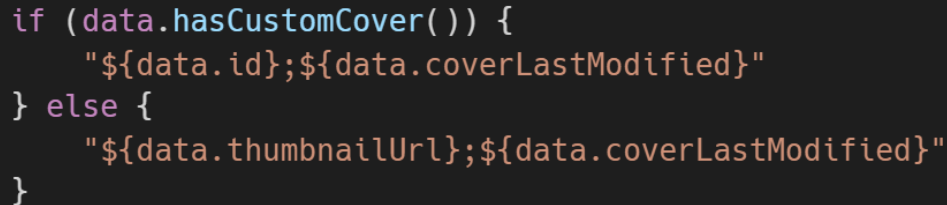
In addition to the efficient energy management during screen transitions and events, the Tachiyomi app exhibits a judicious use of internet resources. The application minimizes unnecessary data fetches, optimizing its internet consumption. By strategically handling data retrieval processes, the app ensures that network requests are streamlined and focused, contributing to a responsive and energy-efficient user experience. This dual emphasis on both energy and internet efficiency reflects the app's commitment to providing users with a smooth and resource-conscious manga reading experience. Notably, only one significant peak is identified throughout the entire scenario, occurring when applying filters to read manga.

### **Micro Optimization**

#### **Micro-optimization Strategies:**

## 1. Caching Custom Covers:

- **Micro-optimization:**

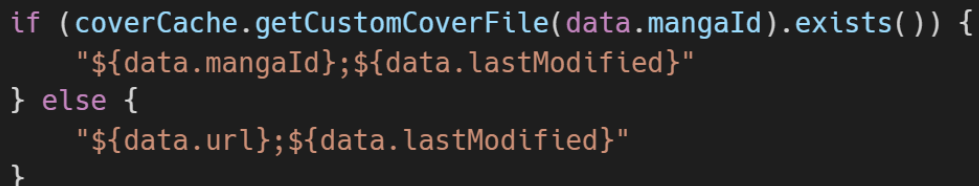


```
if (data.hasCustomCover()) {  
    "${data.id};${data.coverLastModified}"  
} else {  
    "${data.thumbnailUrl};${data.coverLastModified}"  
}
```

- **Location:** Found in `MangaKeyer` class in the `key` function.
- **Reason:** This is a micro-optimization as it optimizes the key generation process for Coil image loading library. If a manga has a custom cover, it uses the manga ID along with the cover's last modified timestamp as the key. Otherwise, it uses the thumbnail URL along with the last modified timestamp. This ensures unique keys for custom and non-custom covers, improving cache utilization.
- **Purpose:** The purpose is to distinguish between manga with custom covers and those without, ensuring that Coil's image cache is used efficiently by providing distinct keys based on the cover type.

## 2. Custom Cover File Existence Check:

- **Micro-optimization:**



```
if (coverCache.getCustomCoverFile(data.mangaId).exists()) {  
    "${data.mangaId};${data.lastModified}"  
} else {  
    "${data.url};${data.lastModified}"  
}
```

- **Location:** Located in the `MangaCoverKeyer` class in the `key` function.

- **Reason:** This is a micro-optimization as it optimizes key generation based on the existence of a custom cover file. If the file exists, it uses the manga ID along with the cover's last modified timestamp as the key. Otherwise, it uses the cover URL along with the last modified timestamp. This avoids unnecessary file I/O operations and ensures that the correct key is generated based on the cover's availability.
- **Purpose:** The purpose is to efficiently check for the existence of a custom cover file before generating a key, reducing unnecessary file system access and improving key generation speed.

### 3. FastScrollLazyVerticalGrid:

- **Micro-optimization:**



```
internal fun LazyLibraryGrid(
    modifier: Modifier = Modifier,
    columns: Int,
    contentPadding: PaddingValues,
    content: LazyGridScope() -> Unit,
) {
    FastScrollLazyVerticalGrid(
        columns = if (columns == 0) GridCells.Adaptive(128.dp) else GridCells.Fixed(columns),
        modifier = modifier,
        contentPadding = contentPadding + PaddingValues(8.dp),
        verticalArrangement = Arrangement.spacedBy(CommonMangaItemDefaults.GridVerticalSpacer),
        horizontalArrangement = Arrangement.spacedBy(CommonMangaItemDefaults.GridHorizontalSpacer),
        content = content,
    )
}
```

- **Location:** Located in the LazyLibraryGrid file, where FastScrollLazyVerticalGrid is used instead of the standard LazyVerticalGrid.
- **Reason:** This is a micro-optimization focusing on scrolling performance. The FastScrollLazyVerticalGrid is likely optimized for efficient scrolling behavior, providing a smoother user experience, especially in scenarios with a large number of items.
- **Purpose:** The purpose is to enhance the scrolling experience within the LazyGrid by utilizing a specialized implementation (FastScrollLazyVerticalGrid) that is designed for improved performance.



#### 4. Dispatcher Optimization:

- **Micro-optimization:**

```
class AndroidDatabaseHandler(  
    val db: Database,  
    private val driver: SqlDriver,  
    val queryDispatcher: CoroutineDispatcher = Dispatchers.IO,  
    val transactionDispatcher: CoroutineDispatcher = queryDispatcher,  
)
```

- **Location:** Located in the AndroidDatabaseHandler class.
- **Reason:** This is considered a micro-optimization as it allows for the fine-tuning of coroutine execution contexts, ensuring that certain operations are dispatched on appropriate threads. It optimizes resource utilization and responsiveness.
- **Purpose:** The purpose is to optimize coroutine execution by providing specific dispatchers (Dispatchers.IO by default) for query and transaction operations, enhancing concurrency and minimizing thread contention.

#### 5. ThreadLocal Optimization:

- **Micro-optimization:**

```
val suspendingTransactionId = ThreadLocal<Int>()
```

- **Location:** Located in the AndroidDatabaseHandler class.
- **Reason:** It's a micro-optimization because it allows for thread-local storage of transaction IDs, avoiding conflicts in multi-threaded scenarios. Each thread has its own suspendingTransactionId.
- **Purpose:** The purpose is to ensure that suspending transactions have a unique identifier per thread, preventing interference between transactions executed on different threads.

**Potential Optimization:**

- Ensure that the ``coverCache.getCustomCoverFile(data.mangald)`` operation is efficiently implemented to minimize file system access, as frequent file I/O operations can impact performance.
- Optimize the ``coverCache.get()`` operation to efficiently retrieve cover images from the cache, potentially implementing caching strategies like memory caching for quicker access.
- If there are complex UI components within the LazyGrid items, it will be a good idea to optimize their rendering logic, such as using `remember` for expensive calculations or utilizing `Modifier.drawWithCache` for caching.
- Customize coroutine dispatchers for `queryDispatcher` and `transactionDispatcher` based on specific application needs.

## Audit Report

We, Daniel Bernal and Daniel Gómez, are key members of our company, tasked with the meticulous analysis of Android mobile applications and reporting any technical issues identified in the source code. In this comprehensive audit, we present a thorough analysis of Tachiyomi, a leading manga reading application renowned for its robust feature set, extensive customization options, and a vibrant user community. Our primary focus centers on evaluating the application's performance in critical areas such as GPU rendering, memory management, overdraw prevention, threading strategies, and micro-optimization.

Tachiyomi exhibits exemplary GPU rendering practices by leveraging the power of Kotlin coroutines and implementing a fixed-size thread pool. This strategic approach ensures efficient image loading and seamless network operations, contributing to an optimal user interface. The application's commitment to efficient rendering processes stands out as a cornerstone of its performance.

The application's memory management prowess is evident in its sophisticated caching strategies. Tachiyomi's adoption of a cache-first approach for manga images minimizes unnecessary network requests, showcasing a commitment to resource-efficient operations. The recommended focus on comprehensive error handling aligns with best practices, ensuring robust memory management and a resilient user experience.

Tachiyomi takes a cautious stance on image rendering, minimizing overdraw by prioritizing locally stored data over redundant network requests. This deliberate approach reflects a keen understanding of the importance of optimized rendering layers in preventing visual artifacts. The suggested detailed analysis of rendering layers aims to further refine this aspect for enhanced efficiency.

Threading in Tachiyomi is a strength, with effective implementation of Kotlin coroutines and a focus on user preferences and lifecycle management. The application's responsiveness owes much to this threading strategy, providing a smooth and dynamic user interface. Regular performance monitoring and staying updated with threading best practices are recommended to sustain and enhance this performance.

Micro-optimization strategies, such as dispatcher customization and thread-local storage for transaction IDs, showcase Tachiyomi's commitment to enhancing concurrency. The proposed customization options for coroutine dispatchers and regular reviews of micro-optimizations align with a forward-looking approach to evolving performance needs.

In conclusion, Tachiyomi stands as a benchmark for performance in the manga reading application landscape. While it demonstrates robust performance in GPU rendering, memory management, overdrawing prevention, threading strategies, and micro-optimization, the provided recommendations aim to elevate its performance further. Implementing these suggestions will not only solidify Tachiyomi's reputation for reliability and responsiveness but also ensure its continued delivery of an optimal manga reading experience. Regular performance monitoring, adherence to best practices, and an agile approach to evolving needs will be instrumental in maintaining Tachiyomi's performance excellence.