Mobile App Performance Profiling and Optimization

Practical Exercise – Performance Profiling and Optimization

Introduction to App Efficiency Measurement

Course: Mobile Application Development (Chapter 8)
Level: Technical University Students
Duration: 7–9 hours self-study + 1.5–2 hours online discussion
Format: Individual Assignment with Performance Analysis (5–6 pages)

Exercise Overview

In this exercise, you will master comprehensive performance profiling methodologies for mobile applications: understanding metrics for measuring app efficiency, analyzing tools for speed and memory profiling, optimizing power consumption and responsiveness, and handling critical scenarios including graphics rendering, animations, background processing, and multithreaded execution. You will learn to identify performance bottlenecks, measure their impact on user experience, and implement targeted optimization strategies across the full application stack[1][2].

Key Learning Outcomes

- Understand performance profiling methodologies and key metrics (FPS, memory, CPU, power)
- Master platform-specific profiling tools (Android Profiler, Xcode Instruments, DevTools)
- Analyze and optimize app startup time, memory usage, and battery consumption
- Profile and optimize graphics rendering and animation performance
- Implement efficient background processing and asynchronous operations
- Design and execute multithreaded execution patterns safely
- Identify and resolve memory leaks, jank, and responsiveness issues
- Conduct performance testing and establish performance baselines

---

Part A: Performance Profiling Fundamentals

Task 1: Understanding Performance Metrics and Profiling Tools (Self-Study: 2–2.5 hours)

Objective: Master the core performance metrics and profiling tools available for mobile applications.

Instructions:

1. **Define Key Performance Metrics:**

Create detailed definitions (2–3 sentences each) for the following metrics:

Frames Per Second (FPS):
FPS measures how many complete frames the application renders per second, with 60 FPS considered the gold standard for smooth user interaction on most devices. When FPS drops below 50, users perceive jank and stuttering. Consistent 60 FPS (or 120 FPS on high-refresh devices) is essential for scrolling, animations, and interactive gestures[1].

Memory Footprint:
Memory footprint refers to the total amount of RAM consumed by the application, including heap memory (objects), native memory (C/C++ allocations), and graphics memory (GPU). Excessive memory usage (>500 MB) causes crashes on lower-end devices and forces garbage collection, contributing to jank. Baseline memory should typically stay under 100 MB for standard apps[2].

CPU Usage:
CPU usage represents the percentage of processor utilization by the application across all cores. High sustained CPU usage (>80%) drains battery quickly and impacts other applications. Profiling identifies CPU-intensive operations that should run on background threads or use native optimization[1].

Power Consumption:
Power consumption measured in watts or milliwatt-hours (mWh) indicates battery drain rate. Mobile applications should consume <50 mW during idle states and optimize wake-lock usage. Excessive power consumption may indicate unnecessary background processing, constant network access, or inefficient sensor polling[2].

Responsiveness / Input Latency:
Responsiveness (typically measured in milliseconds) indicates the delay between user input and visible app response. Touch-to-screen latency should be <100 ms for perceived responsiveness. Jank during interactions (>16 ms per frame for 60 FPS) indicates main thread blocking[1].

Startup Time:
Startup time measures duration from app launch to first user interaction. Cold startup (full app initialization) typically takes 2–5 seconds; warm startup (from background) should be <1 second. Excessive startup time indicates heavy initialization, blocking I/O, or large asset loading[2].

Garbage Collection Pauses:
Garbage collection (GC) pauses occur when the runtime reclaims unused memory, temporarily freezing application execution. Pauses >50 ms become noticeable to users as jank. Frequent GC indicates excessive object allocation or memory leaks[1].

2. **Platform-Specific Profiling Tools:**

Android Profiling Tools:

Tool	Purpose	Key Metrics
Android Profiler	Real-time CPU, memory, network profiling	CPU %, RAM, Jank
Layout Inspector	Visual hierarchy and layout debugging	Component bounds, margins
Systrace	System-level tracing and analysis	Frame timing, thread interaction
Memory Profiler	Heap analysis and memory leak detection	Allocations, GC events, leaks
CPU Profiler	Method-level profiling	Call stacks, execution time
Network Profiler	Network request analysis	Bandwidth, request timing

Table 1: Android Profiling Tools Overview

iOS Profiling Tools:

Tool	Purpose	Key Metrics
Xcode Instruments	Real-time performance profiling	CPU, memory, I/O
Core Animation	Graphics rendering optimization	Frame rate, GPU utilization
Allocations	Memory allocation tracking	Heap size, object counts
Leaks	Automatic memory leak detection	Leaked objects, locations
System Trace	Multi-threaded execution analysis	Thread scheduling, locks
Energy Impact	Battery consumption analysis	Power usage, wake-locks

Table 2: iOS Profiling Tools Overview

React Native / Flutter Cross-Platform Tools:

- **React DevTools**: Performance profiler for React component renders
- **Flutter DevTools**: Real-time frame rendering, memory profiling
- **Sentry**: Real-world performance monitoring and crash reporting
- **Firebase Performance Monitoring**: App-level metrics collection

- **NewRelic Mobile**: Comprehensive mobile monitoring
- **DataDog Mobile**: Performance and error tracking

3. **Profiling Workflow Example:**
4. Establish Baseline Metrics
    - Run app on target device (representative of user devices)
    - Measure FPS during common workflows (scrolling, navigation)
    - Record memory baseline (cold start, after 5 min, after navigation)
    - Document startup time (cold and warm)
5. Identify Bottlenecks
    - Run app under profiler while executing user workflows
    - Record CPU usage patterns
    - Capture memory allocation spikes
    - Analyze frame drops and jank occurrences
6. Prioritize Issues
    - Rank issues by user impact (jank visible to users > background inefficiency)
    - Estimate optimization effort vs. performance gains
    - Focus on high-impact, achievable improvements
7. Implement Optimizations
    - Apply targeted fixes for top bottlenecks
    - Measure impact after each optimization
    - Compare to baseline metrics
8. Validate Results
    - Run profiler again after optimizations
    - Verify improvements meet targets
    - Test on multiple devices (high-end and low-end)
    - Conduct battery and thermal testing

Deliverable:

- Definitions of 7 key performance metrics (2–3 sentences each)
- Completed profiling tools comparison tables (Android, iOS)
- Profiling workflow diagram or description
- 2–3 sentence reflection on metric priorities

Part B: Graphics Rendering and Animation Optimization

Task 2: Rendering Performance and Animation Analysis (Self-Study: 2.5–3 hours)

Objective: Master graphics rendering optimization and smooth animation implementation across platforms.

Instructions:

1. **Graphics Rendering Pipeline:**

Understand the rendering pipeline from code to pixels:

Application Code
↓
Layout/Measurement Phase (CPU: calculate view sizes)
↓
Draw Phase (CPU: generate drawing commands)
↓
Rendering Phase (GPU: rasterize to framebuffer)
↓
Display (Vsync: display to screen @ 60 Hz)

Each phase must complete within the frame budget (~16.67 ms for 60 FPS):

- **Measure Phase** (0–8 ms): Calculate layout dimensions
- **Layout Phase** (0–8 ms): Position views
- **Draw Phase** (0–8 ms): Generate GPU commands
- **GPU Rendering** (parallel to phases above): Execute GPU work
- **Vsync Synchronization** (at 60 Hz): Display to screen

Any phase exceeding ~16 ms causes the frame to drop, resulting in noticeable jank[1][2].

2. **Common Rendering Bottlenecks:**

Slow Layout/Measure Phase:

// ❌ PROBLEMATIC: Deep nested layouts with complex constraints

// ☑ OPTIMIZED: Flatter hierarchy

Slow Draw Phase:

// ❌ PROBLEMATIC: Creating new objects during draw
override fun onDraw(canvas: Canvas) {
val paint = Paint() // NEW object every frame!

```
canvas.drawCircle(100f, 100f, 50f, paint)
}

// ☑ OPTIMIZED: Reuse objects
private val paint = Paint()
override fun onDraw(canvas: Canvas) {
canvas.drawCircle(100f, 100f, 50f, paint)
}
```

Heavy GPU Operations:

```
// ❌ PROBLEMATIC: Rendering shadows every frame


// ☑ OPTIMIZED: Bake shadow into image
```

3. **Animation Performance Optimization:**

Property Animations (Efficient):

Property animations modify object properties without invalidating the entire view hierarchy:

```
// ☑ EFFICIENT: ObjectAnimator modifies single property
ObjectAnimator.ofFloat(view, "translationX", 0f, 100f).apply {
duration = 300
start()
}

// GPU-accelerated properties: translationX, translationY, scaleX, scaleY, alpha, rotation
```

View Animations (Less Efficient):

View animations create transformation matrices but don't update the actual layout:

```
// ⚠ LESS EFFICIENT: Applies transformation, doesn't update layout
val animation = TranslateAnimation(0f, 100f, 0f, 0f)
animation.duration = 300
view.startAnimation(animation)
// View appears moved, but layout remains at original position
```

Drawing Animations (Inefficient):

Frame-by-frame invalidation and redrawing:

```
// ❌ INEFFICIENT: Custom Canvas drawing every frame
override fun onDraw(canvas: Canvas) {
canvas.drawRect(x, y, x + 100, y + 100, paint)
}
// Triggers layout, measure, draw phases every frame
```

4. **Smooth Animation Implementation:**

Flutter Animation Example:

```
class AnimatedContainerExample extends StatefulWidget {
@override
State<AnimatedContainerExample> createState() => _AnimatedContainerExampleState();
}

class _AnimatedContainerExampleState extends State<AnimatedContainerExample>
with SingleTickerProviderStateMixin {
late AnimationController _controller;
late Animation<Offset> _animation;

@override
void initState() {
super.initState();
_controller = AnimationController(
duration: const Duration(milliseconds: 500),
vsync: this,
);

_animation = Tween<Offset>(
  begin: Offset.zero,
  end: const Offset(1, 0),
).animate(CurvedAnimation(parent: _controller, curve:
Curves.easeInOut));

}

@override
Widget build(BuildContext context) {
return SlideTransition(
position: _animation,
child: Container(
width: 100,
height: 100,
color: Colors.blue,
),
);
}

@override
void dispose() {
_controller.dispose();
super.dispose();
}
}
```

React Native Animation Example:

```
import { Animated, Easing } from 'react-native';

class AnimatedComponent extends React.Component {
constructor(props) {
super(props);
this.animatedValue = new Animated.Value(0);
}
```

```
startAnimation = () => {
Animated.timing(this.animatedValue, {
toValue: 100,
duration: 500,
easing: Easing.inOut(Easing.ease),
useNativeDriver: true, // GPU acceleration
}).start();
};

render() {
const translateX = this.animatedValue.interpolate({
inputRange: [0, 100],
outputRange: ['0%', '100%'],
});

return (
  <Animated.View style={{ transform: [{ translateX }] }}>
    <View style={{ width: 100, height: 100, backgroundColor: 'blue' }}
/>
  </Animated.View>
);

}
}
```

5. **Rendering Optimization Checklist:**

- Flatten view hierarchies (target <10 depth for complex screens)
- Use hardware acceleration judiciously (enabledZ property, canvas rendering)
- Reuse Paint, Bitmap, and drawable objects
- Profile with platform tools (Core Animation, Systrace)
- Use property animations and GPU-accelerated properties
- Implement rendering caches for complex drawings
- Avoid invalidating large view hierarchies
- Test animations on low-end devices
- Use VSync-aware rendering
- Profile frame timing and identify jank sources

Deliverable:

- Rendering pipeline explanation with frame budget breakdown
- Identified common bottlenecks with code examples
- Animation implementation comparison (efficient vs. inefficient)
- Complete animation code example (Flutter or React Native)
- Performance optimization checklist with justifications

Part C: Memory Management and Background Processing

Task 3: Memory Profiling and Background Task Optimization (Self-Study: 2–2.5 hours)

Objective: Master memory leak detection, efficient memory management, and background processing strategies.

Instructions:

1. **Memory Profiling Workflow:**

Step 1: Capture Baseline

1. Start profiler (Memory Profiler, Allocations tool)
2. Launch app (cold start)
3. Wait for initialization to complete
4. Record memory snapshot
    - Heap size: ~50-100 MB
    - Native size: ~10-30 MB
    - Graphics: ~20-50 MB
5. Document baseline

Step 2: Execute User Workflows

1. Scroll through lists (10-20 seconds)
2. Navigate between screens (5-10 times)
3. Load images and media (10-20 items)
4. Monitor memory growth
    - Should grow <20 MB during workflow
    - Should stabilize after workflow
5. Document memory after each action

Step 3: Identify Leaks

1. Perform same workflow multiple times
2. Trigger garbage collection between cycles
3. Compare heap sizes:
    - Memory should return to near-baseline
    - Growing memory ≠ leak (may be caches)
    - Memory NOT returning = potential leak
4. Use heap snapshots to identify retained objects

Step 4: Analyze Allocations

1. Enable allocation tracking
2. Execute workflow
3. Examine allocation tree:
    - Count: how many objects
    - Size: memory per object
    - Location: allocation stack trace
4. Identify allocation hotspots
    - In tight loops: move outside loop
    - Large objects: consider pooling
    - Frequent creation: cache or reuse
5. **Common Memory Leaks:**

Listener/Callback Leaks:

// ❌ LEAK: Activity reference held by listener
class MyActivity : AppCompatActivity() {
private val listener = object : OnDataChangeListener {
override fun onChange(data: Any) {
this@MyActivity.updateUI(data) // Holds Activity reference
}
}

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    EventBus.register(listener) // Never unregistered!
}
```

}

// ☑ FIXED: Unregister listener on destroy
override fun onDestroy() {
super.onDestroy()
EventBus.unregister(listener)
}

Static Reference Leaks:

// ❌ LEAK: Static reference to Activity
class Singleton {
companion object {
var activity: Activity? = null
}
}

```
class MyActivity : AppCompatActivity() {
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
Singleton.activity = this // Static reference persists!
}
}

// ☑ FIXED: Use WeakReference or avoid static references
class Singleton {
companion object {
var activity: WeakReference<Activity>? = null
}
}

Singleton.activity = WeakReference(this)
```

Circular Reference Leaks:

```
// ❌ LEAK: Parent → Child → Parent circular reference
class Parent {
val child = Child(this)
}

class Child(val parent: Parent) {
val reference = parent // Circular reference prevents GC
}

// ☑ FIXED: Use WeakReference for back-references
class Child(val parent: Parent) {
val reference = WeakReference(parent)
}
```

Coroutine and Thread Leaks:

```
// ❌ LEAK: Coroutine holds Activity reference
class MyActivity : AppCompatActivity() {
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
lifecycleScope.launch {
delay(10000) // 10 second delay
// Activity may be destroyed during this delay
updateUI() // Still tries to update destroyed Activity
}
}
}

// ☑ FIXED: Use lifecycleScope.launch to auto-cancel
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
lifecycleScope.launchWhenCreated { // Auto-cancels on destroy
delay(10000)
updateUI()
}
}
```

3. **Background Processing Strategies:**

WorkManager (Android):

```
import androidx.work.Worker
import androidx.work.WorkerParameters
import androidx.work.PeriodicWorkRequestBuilder
import java.util.concurrent.TimeUnit

class DataSyncWorker(context: Context, params: WorkerParameters) : Worker(context,
params) {
override fun doWork(): Result {
return try {
// Perform background work
val data = fetchDataFromServer()
saveToDatabase(data)
Result.success()
} catch (e: Exception) {
// Retry on failure
Result.retry()
}
}
}

// Schedule periodic work
val syncWork = PeriodicWorkRequestBuilder<DataSyncWorker>(
15, TimeUnit.MINUTES // Repeat every 15 minutes
).build()

WorkManager.getInstance(context).enqueueUniquePeriodicWork(
"data_sync",
ExistingPeriodicWorkPolicy.KEEP,
syncWork
)
```

Flutter Background Execution:

```
import 'package:flutter_background_service/flutter_background_service.dart';

void startBackgroundService() {
final service = FlutterBackgroundService();

service.startService();

service.onDataReceived.listen((event) {
// Handle data from background service
});
}

// Background task
Future<void> backgroundTask() async {
// Long-running operation
final result = await performDataSync();
```

```
// Send result to foreground
FlutterBackgroundService().invoke('update_data', result);
}
```

Push Notification Handling:

```
// ❌ INEFFICIENT: Wake lock held entire message duration
val wakeLock = PowerManager.newWakeLock(
PowerManager.PARTIAL_WAKE_LOCK,
"messaging:wakelock"
)
wakeLock.acquire() // Acquired indefinitely

// ☑ EFFICIENT: Release immediately after processing
wakeLock.acquire(30000) // 30 second timeout
try {
handlePushMessage(message)
} finally {
wakeLock.release()
}
```

4. **Memory Optimization Best Practices:**

- Use lazy initialization for heavy objects
- Implement object pooling for frequently created objects
- Use weak references for cache entries and listeners
- Profile with memory tools regularly during development
- Monitor memory across device configurations
- Implement proper lifecycle management
- Use scoped coroutines (lifecycleScope, viewModelScope)
- Limit bitmap sizes; scale down if oversized
- Clear references in onDestroy() and onDestroyView()
- Avoid static references to Context or Activities
- Use background execution frameworks (WorkManager, etc.)
- Test memory behavior on low-end devices
- Implement memory warnings and graceful degradation

Deliverable:

- Memory profiling workflow with step-by-step instructions
- Three identified memory leaks with code examples and fixes
- Background processing implementation (WorkManager + Flutter example)
- Memory optimization checklist with justifications

Part D: Multithreaded Execution and Responsiveness

Task 4: Threading, Concurrency, and Main Thread Optimization (Self-Study: 1.5–2 hours)

Objective: Master safe multithreading patterns and main thread optimization for responsiveness.

Instructions:

1. **Threading Architecture:**

Most mobile frameworks use single-threaded UI model:

Main Thread (UI Thread)
├─ Layout & Measurement
├─ Drawing & Rendering
├─ Event Handling (Touch, Lifecycle)
└─ ❌ NO LONG-RUNNING OPERATIONS

Background Threads
├─ Network I/O
├─ Database Operations
├─ File I/O
├─ Computationally Intensive Tasks
└─ Heavy Data Processing

Golden Rule: Never block the main thread. Any operation >16 ms on main thread causes jank[1][2].

2. **Main Thread Blocking Examples:**

Network Request on Main Thread:

// ❌ BLOCKS MAIN THREAD: Network call
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)

```
val response = api.fetchUser() // ~500 ms network latency
displayUser(response) // Won't execute for 500 ms
```

}

// ☑ FIXED: Move to background thread
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)

```
Thread {
    val response = api.fetchUser() // Background thread
    runOnUiThread {
```

```
        displayUser(response) // Back to main thread for UI
    }
}.start()
```

}

// ☑ BETTER: Use coroutines
lifecycleScope.launch {
val response = withContext(Dispatchers.IO) {
api.fetchUser() // Background thread
}
displayUser(response) // Main thread automatically
}

Database Query on Main Thread:

// ✕ BLOCKS MAIN THREAD
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
super.onViewCreated(view, savedInstanceState)

```
val users = database.userDao().getAllUsers() // Disk I/O
userAdapter.setUsers(users)
```

}

// ☑ FIXED: Room + Coroutines
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
super.onViewCreated(view, savedInstanceState)

```
viewLifecycleOwner.lifecycleScope.launch {
    val users = userViewModel.users.collect { userList ->
        userAdapter.setUsers(userList)
    }
}
```

}

Image Processing on Main Thread:

// ✕ BLOCKS MAIN THREAD: Complex image manipulation
bitmap = Bitmap.createBitmap(originalBitmap).apply {
for (x in 0 until width) {
for (y in 0 until height) {
// Pixel manipulation: ~1000 ms for 1000x1000 image
val pixel = getPixel(x, y)
// Complex processing...
}
}
}

// ☑ FIXED: Async processing
Thread {

bitmap = processImage(originalBitmap)
runOnUiThread {
imageView.setImageBitmap(bitmap)
}
}.start()

3. **Safe Concurrency Patterns:**

Kotlin Coroutines (Recommended):

// Safe concurrent operations with automatic thread management
viewModel.viewModelScope.launch {
try {
// Execute on IO thread
val users = withContext(Dispatchers.IO) {
database.userDao().getAllUsers()
}

```
    // Switch to main thread
    withContext(Dispatchers.Main) {
        displayUsers(users)
    }
} catch (e: Exception) {
    // Exception handling
    showError(e.message)
}
```

}

Thread Pools (Java/Android Traditional):

val executor = Executors.newFixedThreadPool(4)

executor.execute {
// Background work
val result = performHeavyCalculation()

```
// Post back to main thread
handler.post {
    updateUI(result)
}
```

}

Flutter Isolates (Multi-threading):

import 'dart:isolate';

// Heavy computation in separate isolate
Future<void> computeHeavyTask() async {
final result = await compute(heavyCalculation, inputData);
setState(() {
resultData = result;

```
});
}

int heavyCalculation(int data) {
// Expensive computation, runs in separate thread
int sum = 0;
for (int i = 0; i < 1000000000; i++) {
sum += i;
}
return sum + data;
}
```

4. **Input Responsiveness Optimization:**

Respond to Touch Input Quickly:

```
// ❌ SLOW: Heavy work in touch handler
override fun onTouchEvent(event: MotionEvent): Boolean {
when (event.action) {
MotionEvent.ACTION_DOWN -> {
// Heavy processing: ~200 ms
calculateComplexLayout()
invalidate()
}
}
return true
}

// ☑ FAST: Immediate visual feedback, work async
override fun onTouchEvent(event: MotionEvent): Boolean {
when (event.action) {
MotionEvent.ACTION_DOWN -> {
// Immediate visual feedback
showPressedState()

        // Defer heavy work
        Thread {
            calculateComplexLayout()
            runOnUiThread { invalidate() }
        }.start()
    }
}
return true

}
```

Debounce Expensive Operations:

```
// ❌ INEFFICIENT: Search triggers on every key
searchEditText.addTextChangedListener(object : TextWatcher {
override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
performSearch(s.toString()) // 100 queries for 100 characters
}
override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {}
```

```
override fun afterTextChanged(s: Editable?) {}
})

// ☑ EFFICIENT: Debounce with delay
private var searchJob: Job? = null
searchEditText.addTextChangedListener(object : TextWatcher {
override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
searchJob?.cancel()
searchJob = viewLifecycleOwner.lifecycleScope.launch {
delay(300) // Wait 300 ms for user to stop typing
performSearch(s.toString()) // Single query
}
}
override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {}
override fun afterTextChanged(s: Editable?) {}
})
```

5. **Responsive Metrics and Targets:**

Interaction	Target	Priority
Touch input response	< 100 ms	Critical
Scroll response	< 16 ms per frame	Critical
Button tap feedback	< 200 ms	High
Navigation transition	200–300 ms	High
Search results display	< 500 ms	Medium
Image loading (UI ready)	< 1000 ms	Medium

Table 3: Interaction Latency Targets

Deliverable:

- Threading architecture diagram with main/background thread responsibilities
- Three main thread blocking examples with fixes
- Safe concurrency implementation (Coroutines + Isolates)
- Responsiveness optimization code examples
- Responsive metrics targets table

Part E: Comprehensive Performance Analysis and Optimization Report

Task 5: Performance Analysis and Strategic Optimization (Self-Study: 1.5–2 hours)

Objective: Conduct complete performance analysis and develop optimization strategy.

Instructions:

1. **Case Study: Social Feed Application**

Initial Performance Baseline:

Metric	Current	Target
Startup Time (Cold)	5.2 seconds	2.5 seconds
Startup Time (Warm)	1.8 seconds	1.0 second
Memory (Baseline)	180 MB	100 MB
Memory (After loading 50 posts)	280 MB	120 MB
Feed Scroll FPS	45-52 FPS	55-60 FPS
Image Load Time	1.5 seconds	500 ms
Battery Drain (30 min usage)	12%	3%

Table 4: Initial Performance Baseline

Identified Bottlenecks:

1. **High startup time**: Loading entire database of posts on launch
2. **Memory growth**: No image caching strategy
3. **Jank during scroll**: Complex post layouts with shadows
4. **Battery drain**: Constant network polling in background
5. **Image loading**: Downloading full resolution images

Optimization Strategy:

- **Startup**: Implement lazy loading; load top 20 posts only
- **Memory**: Add image caching with LRU eviction
- **Rendering**: Simplify post layout; reduce shadow rendering
- **Battery**: Implement intelligent background sync (WiFi only, longer intervals)
- **Images**: Implement progressive image loading (thumbnail → full res)

Expected Improvements:

Metric	Before	After	Improvement
Startup Time (Cold)	5.2s	2.3s	56%
Startup Time (Warm)	1.8s	0.9s	50%
Memory (Baseline)	180 MB	95 MB	47%
Memory (After loading 50 posts)	280 MB	115 MB	59%

Feed Scroll FPS	48 FPS	58 FPS	21%
Image Load Time	1.5s	0.6s	60%
Battery Drain (30 min usage)	12%	4%	67%

Table 5: Optimization Results

2. **Optimization Implementation Details:**

Lazy Loading Implementation (Kotlin):

class PostViewModel : ViewModel() {
private val _posts = MutableLiveData<List<Post>>()
val posts: LiveData<List<Post>> = _posts

```
private var offset = 0
private val pageSize = 20

fun loadInitialPosts() {
    viewModelScope.launch {
        val initialPosts = withContext(Dispatchers.IO) {
            postRepository.getPosts(offset = 0, limit = pageSize)
        }
        _posts.value = initialPosts
        offset = pageSize
    }
}

fun loadMorePosts() {
    viewModelScope.launch {
        val morePosts = withContext(Dispatchers.IO) {
            postRepository.getPosts(offset = offset, limit = pageSize)
        }
        _posts.value = (_posts.value ?: emptyList()) + morePosts
        offset += pageSize
    }
}
```

}

Image Caching Strategy:

import androidx.lifecycle.ViewModelProvider
import coil.load
import coil.request.CachePolicy
import coil.request.ImageRequest

// Use Coil for efficient image caching
imageView.load(imageUrl) {
crossfade(true)
memoryCachePolicy(CachePolicy.ENABLED)
diskCachePolicy(CachePolicy.ENABLED)

```
// Load placeholder while downloading
placeholder(R.drawable.image_placeholder)

// Progressive image: thumbnail first
val thumbnailRequest = ImageRequest.Builder(context)
    .data(thumbnailUrl)
    .size(150, 150)
    .build()

ImageView.load(thumbnailRequest)
```

}

Simplified Post Layout (XML):

<View with elevation, shadows, complex nesting.../>

<LinearLayout android:layout_width="match_parent"
android:layout_height="wrap_content" android:orientation="vertical"
android:padding="16dp">

```
<ImageView
    android:id="@+id/post_image"
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:scaleType="centerCrop"/>

<TextView
    android:id="@+id/post_title"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="18sp"
    android:textStyle="bold"
    android:layout_marginTop="8dp"/>

<TextView
    android:id="@+id/post_content"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="4dp"/>
```

</LinearLayout>

Smart Background Sync:

// Only sync on WiFi, at intervals, when battery >20%
val syncConstraints = Constraints.Builder()
.setRequiredNetworkType(NetworkType.CONNECTED)
.setRequiresBatteryNotLow(true)
.build()

```
val syncWork = PeriodicWorkRequestBuilder<PostSyncWorker>(
60, TimeUnit.MINUTES // Longer interval
)
.setConstraints(syncConstraints)
.build()

WorkManager.getInstance(context).enqueueUniquePeriodicWork(
"post_sync",
ExistingPeriodicWorkPolicy.KEEP,
syncWork
)
```

3. **Performance Testing and Validation:**

Before and After Profiling:

```
Cold Startup (Before Optimization):
├─ Database load: 1800 ms
├─ Image loading: 1200 ms
├─ Layout calculation: 800 ms
├─ Network sync: 600 ms
└─ Total: 5200 ms

Cold Startup (After Optimization):
├─ Load initial 20 posts: 300 ms
├─ Thumbnail image caching: 150 ms
├─ Simplified layout: 200 ms
├─ Deferred background sync: 0 ms
└─ Total: 2300 ms (56% improvement)
```

4. **Profiling and Validation Checklist:**

- Establish baseline metrics before optimization
- Profile on target device types (low-end, mid-range, flagship)
- Measure each optimization independently
- Document performance gains with before/after metrics
- Use platform profiling tools (Android Profiler, Xcode Instruments)
- Test across network conditions (WiFi, 4G, 3G)
- Monitor battery impact with realistic usage patterns
- Validate that no new issues introduced
- Test on actual devices, not just emulators
- Document optimization techniques for future reference

Deliverable:

- Complete case study with baseline metrics and optimization strategy
- Optimization implementation code examples (3–4 techniques)
- Before/after performance comparison table
- Detailed profiling and validation checklist

Part F: Strategic Thinking and Best Practices

Task 6: Performance Culture and Continuous Optimization (Self-Study: 1 hour)

Objective: Develop strategic approach to performance as ongoing practice.

Instructions:

1. **Performance Budgets:**

Define acceptable performance thresholds:

Component	Budget	Measurement
Cold Startup	< 3.0s	Time from launch to interactive
Warm Startup	< 1.5s	Time from background to interactive
Main Memory	< 100 MB	Baseline at launch
Frame Budget	16 ms	Per-frame rendering
Network Latency	< 500 ms	API response + processing
Battery (1 hour)	< 5%	Real device battery drain

Table 6: Typical Performance Budgets

2. **Performance Monitoring in Production:**

Integrate performance monitoring:

- **Firebase Performance Monitoring**: Automatic metrics collection
- **Sentry**: Error tracking and performance monitoring
- **Crashlytics**: Crash reporting with performance context
- **Custom Analytics**: Business-specific performance tracking
- **User Feedback**: Identify perceived performance issues

3. **Team Best Practices:**

- Profile early, profile often (during active development)
- Make performance a code review criterion
- Test on low-end devices during development
- Include performance tests in CI/CD pipeline
- Monitor production metrics continuously

- Establish performance regressions as bugs
- Share profiling results and optimization techniques
- Schedule regular performance review sessions
- Educate team on common performance anti-patterns
- Celebrate performance improvements

Deliverable:

- Performance budget table with justifications
- Production monitoring implementation plan
- Team best practices document

Evaluation Criteria

Criteria	Excellent (9–10)	Good (7–8)	Acceptable (5–6)	Needs Improvement
Metric Understanding	All metrics deeply understood	6-7 metrics clearly explained	4-5 metrics understood	Shallow knowledge
Profiling Tool Expertise	Multiple tools mastered	2-3 tools thoroughly understood	Basic tool usage	Minimal tool knowledge
Optimization Strategy	Comprehensive, well-prioritized	Good strategy; most factors covered	Some optimizations identified	Limited strategy
Code Quality	Clean, idiomatic implementations	Generally good, some inconsistency	Readable, lacks sophistication	Difficult to follow
Performance Validation	Rigorous testing, metrics-driven	Good testing with metrics	Basic validation	Minimal testing
Threading Safety	All patterns thread-safe	Mostly correct, minor issues	Some threading issues	Significant threading errors
Communication	Clear explanations, excellent detail	Generally clear, good structure	Readable, lacks depth	Disorganized

Practical Application	Real-world applicable solutions	Mostly practical with minor gaps	Basic applicability	Theoretical only

---

Deliverable Checklist

Submit as PDF or DOCX (5–6 pages):

- [ ] Task 1: Performance metrics definitions + profiling tools overview + workflow
- [ ] Task 2: Rendering pipeline explanation + bottlenecks + animation examples
- [ ] Task 3: Memory profiling workflow + leak examples + background processing
- [ ] Task 4: Threading architecture + main thread blocking fixes + concurrency patterns
- [ ] Task 5: Case study with baseline metrics + optimization details + results
- [ ] Task 6: Performance budgets + monitoring plan + team best practices

---

Resources & References

[1] Google. (2024). Android Performance Documentation.
https://developer.android.com/topic/performance

[2] Apple. (2024). iOS Performance Guidelines.
https://developer.apple.com/videos/performance

[3] Google. (2024). Flutter Performance Best Practices. https://flutter.dev/docs/perf

[4] Meta. (2024). React Native Performance Optimization.
https://reactnative.dev/docs/performance

[5] Android Authority. (2024). Memory Profiling and Leak Detection.
https://www.androidauthority.com/memory-profiling

[6] Firebase. (2024). Performance Monitoring for Mobile.
https://firebase.google.com/docs/perf-mod

[7] Google. (2024). Systrace Documentation.
https://developer.android.com/topic/performance/vitals/systrace

[8] Apple. (2024). Xcode Instruments Performance Tools.
https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/

Timeline Guidance

Self-Study (7–9 hours):

- Hours 1–2.5: Task 1 (profiling fundamentals, tools overview)
- Hours 2.5–5.5: Task 2 & 3 (rendering, memory, background processing)
- Hours 5.5–7: Task 4 (threading, responsiveness)
- Hours 7–8: Task 5 (case study, comprehensive analysis)
- Hours 8–9: Task 6 (strategic thinking, best practices)

Online Sessions (1.5–2 hours):

- 0.5 hours: Task 1–2 discussion (profiling methodology)
- 0.5 hours: Task 3–4 demo (memory and threading patterns)
- 0.5 hours: Task 5 case study walkthrough
- 0.5 hour: Q&A and optimization strategy discussion

Extension Tasks (Optional)

1. Implement comprehensive performance monitoring system
2. Profile and optimize existing open-source mobile app
3. Create performance regression testing framework
4. Build performance visualization dashboard
5. Conduct comparative profiling across frameworks (React Native vs Flutter)
6. Implement advanced caching strategy (multi-level, eviction policies)
7. Create thread pool optimization analysis
8. Develop custom profiling tools for domain-specific metrics

---

Assessment Notes

Grading Rubric:

- Profiling Knowledge: 20%
- Optimization Strategy: 25%
- Code Implementation: 25%
- Validation and Testing: 15%
- Communication: 15%

Partial Credit:
Award proportional credit for partially completed tasks, depth of analysis, and code quality regardless of completeness.

Notes for Instructors

This exercise develops practical performance optimization skills:

- Comprehensive profiling methodology understanding
- Platform-specific tool expertise
- Memory leak detection and prevention
- Responsive UI optimization
- Multi-threading safety and patterns
- Production monitoring and analytics

Discussion Prompts:

- "Why does Flutter outperform React Native in rendering?"
- "When should you optimize vs. accept trade-offs?"
- "How do you prioritize competing optimizations?"
- "What role does team expertise play in performance?"
- "How do you establish performance culture?"
- "What hidden performance costs arise from frameworks?"

Common Mistakes:

- Optimizing without profiling data
- Ignoring real-world device constraints
- Assuming all optimizations are worthwhile
- Not validating optimization impact
- Over-optimizing before achieving MVP
- Ignoring battery and thermal impact
- Assuming emulators represent real devices
- Not involving QA in performance testing