T.C.
KIRKLARELİ ÜNİVERSİTESİ
TEKNİK BİLİMLER MESLEK YÜKSEKOKULU
BİLGİSAYAR TEKNOLOJİLERİ BÖLÜMÜ
BİLGİSAYAR PROGRAMCILIĞI PROGRAMI

PERFORMANS OPTİMİZASYONU VE "JANK" AZALTMA ARAŞTIRMA RAPORU

Sevdenur AKTÜRK

1237008801

MOBİL PROGRAMLAMA

Dr. Öğr. Üyesi Nadir SUBAŞI

KIRKLARELİ

12-2025

İÇİNDEKİLER

KISALTMALAR

FPS: Frame Per Second
UI: User Interface
CPU: Central Processing Unit
GPU: Graphics Processing Unit

1.GİRİŞ

Mobil uygulamalarda kullanıcı deneyiminin kalitesi, uygulamanın performansıyla doğrudan ilişkilidir. Günümüzde kullanıcılar, uygulamaların hızlı açılmasını, ekran geçişlerinin akıcı olmasını ve içeriklerin gecikme olmadan yüklenmesini beklemektedir. Bu bağlamda, jank olarak bilinen kare gecikmeleri, modern mobil uygulamalarda çözülmesi gereken önemli bir performans sorunu olarak karşımıza çıkar.

Flutter, tek thread üzerinde çalışan rendering yapısı ve kendi çizim motoru sayesinde yüksek performans hedeflese de yanlış widget kullanımı, optimize edilmemiş liste yapıları veya ağır CPU işlemlerinin ana thread üzerinde çalıştırılması gibi durumlarda performans kayıpları yaşanabilmektedir.

Bu çalışma kapsamında:

- Önce kasıtlı olarak performansı zayıf bir liste yapısı (SlowListPage) oluşturulmuş,
- Ardından optimize edilmiş bir liste yapısı (FastListPage) geliştirilmiş,
- Bu iki yapı Flutter DevTools kullanılarak analiz edilmiş,
- RepaintBoundary, const widget kullanımı, ListView.builder, gecikmeli işlemlerin izole edilmesi gibi tekniklerle iyileştirmeler yapılmıştır.

Bu proje, teorik performans kavramlarının uygulamalı bir örnek üzerinde değerlendirilmesine katkı sağlamaktadır. Böylece öğrenciler, yalnızca performans problemlerini tespit etmeyi değil, bu problemleri çözmek için doğru teknikleri uygulamayı da öğrenmektedir.

2. FLUTTER'DA PERFORMANS VE JANK KAVRAMI

Flutter, UI bileşenlerini çizmek için kendi rendering motoru olan Skia'yı kullanır ve her kareyi yaklaşık 16.6 ms içinde hazırlamayı hedefler. Bu süre, 60 FPS akıcılığını sağlayabilmek için gereklidir. Eğer işlemler bu süreyi aşarsa kare gecikmeleri yaşanır ve kullanıcı ekranda "takılma" hisseder. Bu durum jank olarak tanımlanır.

2.1. Jank Oluşma Nedenleri

- **Ağır CPU işlemlerinin ana thread üzerinde çalıştırılması**
  Büyük listeleri hesaplamak, JSON parse işlemleri veya karmaşık hesaplamalar UI thread'ini bloklayabilir.
- **Widget'ların gereksiz yere rebuild edilmesi**
  Özellikle setState'in yanlış kullanılması veya geniş widget ağaçlarının küçük değişikliklerde bile yeniden oluşturulması performansı etkiler.
- **ListView gibi bileşenlerin yanlış kullanılması**
  Tüm widget'ları aynı anda oluşturan ListView → performans düşüktür.
  Yalnızca görünen elemanları oluşturan ListView.builder → optimize yapı.
- **Repaint Boundary eksikliği**
  Üst seviyede yapılan değişikliklerin alt seviyedeki widget'ları gereksiz yere yeniden çizmesi.
- **Ağır görsel dosyalarının kullanılması**
  Büyük image dosyaları decode sürecinde takılmalara yol açabilir.

2.2. DevTools ile Performans Analizi

Bu projede, iki ekran arasındaki performans farkı aşağıdaki araçlarla incelenmiştir:

- **Performance Timeline**
  Kare süreleri grafik üzerinde gösterilir. Kırmızı çubuklar → 16 ms üzerindeki gecikmeleri işaret eder.
- **Rebuild Profileri**
  Hangi widget'ların ne sıklıkla rebuild olduğu izlenir.
- **Repaint Rainbow**
  Yeniden çizilen widget bölgeleri renkli şekilde belirtilir. Çok sık yanıp sönen alanlar optimize edilmesi gereken bölgelerdir.

2.3. Optimizasyon Tekniklerinin Önemi

Bu projede uygulanan teknikler, jank problemlerinin azaltılmasında doğrudan etkili olmuştur:

- **ListView.builder kullanımı** → Gereksiz widget oluşturmayı ortadan kaldırır.
- **RepaintBoundary** → Sadece gerekli alanların yeniden çizilmesini sağlar.
- **const widget kullanımı** → Build maliyetini düşürür.
- **Ağır işlemleri isolate üzerinde çalıştırma (teorik)** → CPU yükünü UI thread'den alır.

3. FLUTTER DEVTOOLS İLE PERFORMANS ANALİZİ

Bu projede performans değerlendirmesi yapılırken Flutter DevTools ortamındaki farklı analiz araçlarının (Performance, Timeline, Repaint Rainbow, Inspector ve Rebuild Analyzer) kullanıldığı varsayılmıştır. Amaç, uygulamanın oluşturduğu jank'ın nedenlerini gözlemlemek, ağır işlemlerin UI thread üzerindeki etkilerini tespit etmek ve optimize edilmiş sürümdeki iyileşmeleri karşılaştırmalı biçimde açıklamaktır.

Flutter DevTools, hem UI render sürecini hem de CPU yükünü ayrıntılı bir şekilde izlemeye imkân tanıdığı için performans optimizasyonlarında kritik bir role sahiptir.

3.1. Performance (Zaman Çizelgesi) Analizi

Performance sekmesi, uygulamanın ürettiği karelerin zaman çizelgesi üzerinde gösterildiği bir analiz panelidir. Her kare için geçen süre grafik üzerinde bir bar olarak gösterilir ve 16.6 ms sınırını aşan kareler jank olarak kabul edilir.

Yavaş (SlowListPage) sürümde yapılan gözlemler:

- Ağır CPU işlemleri, her liste elemanı oluşturulurken tetiklendiği için kare sürelerinde düzensiz sıçramalar görülmüştür.
- Bazı karelerin **16 ms sınırını aşarak 25–40 ms aralığına çıktığı**, dolayısıyla UI thread'in yeterince hızlı cevap veremediği tespit edilmiştir.
- Timeline üzerinde kırmızı renkli uyarı barları belirerek jank anları görünür hâle gelmiştir.

Optimize (FastListPage) sürümde elde edilen iyileşmeler:

- ListView.builder yalnızca ekranda görünen elemanları oluşturduğu için CPU yükü büyük ölçüde azalmıştır.
- Kare sürelerinin daha istikrarlı ve düşük değerde seyrettiği görülmüştür.
- Timeline üzerinde kırmızı bar sayısının büyük oranda azaldığı, grafik akışının daha düzenli olduğu gözlenmiştir.
- Bu analiz, optimizasyon adımlarının performans üzerinde doğrudan etkili olduğunu açıkça göstermektedir.

3.2. Repaint Rainbow Analizi

Repaint Rainbow, ekranda hangi widget'ların yeniden çizildiğini renklerle gösteren bir görselleştirme aracıdır. Yeniden çizim sayısı arttıkça ekran daha fazla renkle yanıp söner – bu da performans kayıplarının bir göstergesidir.

Yavaş sürümde yapılan gözlemler:

- ListView yapısı optimize edilmediği için scroll işlemi sırasında tüm liste alanı yeniden çizilmiştir.
- Küçük bir hareket bile geniş bir repaint bölgesine neden olmuş; bu da çok sayıda render işlemini tetiklemiştir.
- Repaint Rainbow sürekli ve yoğun şekilde yanıp sönerek performans kaybını net biçimde göstermiştir.

Optimize sürümdeki Repaint iyileştirmeleri:

- RepaintBoundary kullanılarak her satır bağımsız bir çizim alanına dönüştürülmüştür.
- Bu sayede scroll sırasında yalnızca ekrana giren–çıkan widget'lar yeniden çizilmiş, geri kalan alan sabit kalmıştır.
- Repaint bölgeleri belirgin şekilde daralmış, bu da daha akıcı bir kaydırma deneyimi sağlamıştır.

Bu gözlem, RepaintBoundary kullanımının performansa olan olumlu etkisini açık şekilde ortaya koymaktadır.

3.3. Widget Rebuild Analizi

Widget rebuild analizi, hangi widget'ların ne sıklıkla baştan oluşturulduğunu gösterir. Gereksiz rebuild işlemleri performans kayıplarının önemli bir nedenidir.

Yavaş sürümde tespit edilen problemler:

- Normal ListView kullanımı sebebiyle tüm liste elemanları baştan oluşturulmuştur.
- Sabit yapıda olan metin veya container gibi bileşenlerde const kullanılmadığı için gereksiz rebuild işlemleri artmıştır.
- Rebuild sayısındaki artış, CPU yükünü artırarak UI thread'in daha fazla bloklanmasına neden olmuştur.

Optimize sürümde yapılan iyileştirmeler:

- const widget kullanımı, değişmeyen bileşenlerin yeniden oluşturulmasını engellemiştir.

- ListView.builder, yalnızca ekranda görünen elemanları üreterek rebuild yükünü dramatik şekilde azaltmıştır.
- Rebuild Analyzer üzerinde yeniden oluşturulan widget sayısının ciddi oranda düştüğü gözlemlenmiştir.

Sonuç olarak rebuild optimizasyonları, CPU ve render yükünü azaltarak uygulamanın daha kararlı bir performans sergilemesini sağlamıştır.

4. ÖRNEK UYGULAMA SENARYOSU

Bu projede Flutter'ın performans davranışlarını görebilmek amacıyla aynı kullanıcı arayüzünün hem kasıtlı olarak yavaşlatılmış (jank'li) bir sürümü hem de optimize edilmiş (akıcı) bir sürümü oluşturulmuştur.

Bu iki sürüm arasındaki farklar, performans problemlerinin nasıl ortaya çıktığını ve doğru optimizasyon teknikleri uygulandığında nasıl çözümlendiğini gösteren somut bir deney ortamı sağlamaktadır.

4.1. Kasıtlı Olarak Yavaşlatılmış Uygulama (Jank'li Sürüm)

Bu sürüm, Flutter uygulamalarında yaygın olarak karşılaşılan performans problemlerini demonstratif olarak göstermek amacıyla tasarlanmıştır. Jank'in oluşmasına neden olan çeşitli yanlış uygulama örneklerini içermektedir.

Bu sürümün temel özellikleri:

- **Liste boyutu yüksektir.**
  Uygulamada yaklaşık 500 eleman içeren bir liste kullanılmıştır. Bu durum, özellikle yanlış liste yapısı tercih edildiğinde render işlemlerinin artmasına yol açmaktadır.

- **Ağır işlem ana thread üzerinde çalışmaktadır.**
  Her liste elemanı oluşturulurken CPU yoğunluklu bir hesaplama çalıştırılmaktadır. Bu işlem, UI thread'i bloke ederek kare sürelerinin yükselmesine neden olmaktadır.

- **Liste ListView ile statik olarak oluşturulmuştur.**
  ListView.builder yerine klasik ListView kullanılması, tüm liste elemanlarının uygulama başlangıcında oluşturulmasına neden olur. Kullanıcı sadece birkaç elemanı görse dahi tüm liste render edilmektedir.

- **Widget'larda const anahtar kelimesi kullanılmamıştır.**
  Sabit yapılar const ile işaretlenmediğinde Flutter bu widget'ları her yeniden çizimde yeniden oluşturmaktadır. Bu durum gereksiz sayıda rebuild işlemine neden olmaktadır.

- **RepaintBoundary kullanılmamıştır.**
  Listenin tamamı tek bir repaint alanı olarak işlem görmektedir. Böylece yalnızca bir satır değişse bile listenin tamamı yeniden çizilmektedir. Bu da GPU yükünü artırmaktadır.

Genel değerlendirme:

Bu sürüm, performans açısından verimsizdir ve özellikle kaydırma sırasında belirgin jank oluşmaktadır. DevTools Timeline incelemelerinde kare sürelerinin zaman zaman 16 ms

sınırını aşarak 40–80 ms seviyelerine çıktığı gözlemlenmektedir. Bu durum Flutter'ın tek thread üzerinde çalışan UI yapısının performans açısından ne kadar hassas olduğunu göstermektedir.

4.2. Optimizasyon Sonrası Uygulama (Hızlı Sürüm)

Bu sürümde, jank'li yapıda tespit edilen performans problemlerini gidermek amacıyla iyileştirmeler uygulanmıştır. Amaç, kullanıcı deneyimini olumsuz etkileyen gereksiz render ve rebuild işlemlerini ortadan kaldırmak ve daha stabil bir UI elde etmektir.

Bu sürümde uygulanan optimizasyonlar:

- **ListView.builder kullanılmıştır.**
  Bu yapı, yalnızca ekranda görünen elemanların oluşturulmasını sağlar. Bu sayede CPU ve bellek kullanımı önemli ölçüde azaltılmıştır.

- **RepaintBoundary uygulanmıştır.**
  Her bir liste elemanı bağımsız bir çizim alanına alınmış, böylece yalnızca değişiklik olan satırlar yeniden boyanmıştır. Bu yapı GPU üzerindeki yükü azaltmış ve kaydırma performansını artırmıştır.

- **const anahtar kelimesi kullanılmıştır.**
  Sabit widget'ların const olarak işaretlenmesi, gereksiz rebuild işlemlerini azaltmış ve widget ağacını daha stabil hâle getirmiştir.

- **Ağır işlem azaltılmış veya UI thread'den ayrılmıştır.**
  Hesaplama maliyeti düşürülmüş ve gerekli durumlarda işlem yükünün isolate üzerine taşınabileceği değerlendirilmiştir. Bu yaklaşım, UI thread'in akıcılığını korumaktadır.

Genel değerlendirme:

Optimize edilmiş sürümde kaydırma deneyiminin belirgin biçimde iyileştiği gözlenmiştir.

- Kare sürelerinin büyük ölçüde 16 ms altında kaldığı,
- Repaint Rainbow üzerinde yalnızca ilgili satırların yeniden çizildiği,
- Timeline grafiğinde jank oluşumunun ortadan kalktığı
  tespit edilmiştir.

Bu sonuçlar, Flutter uygulamalarında performans sorunlarının doğru tekniklerle kolayca giderilebileceğini göstermektedir.

5. KULLANILAN OPTİMİZASYON TEKNİKLERİ

Bu projede jank oluşumunu azaltmak ve uygulamanın kaydırma performansını iyileştirmek amacıyla dört temel optimizasyon yaklaşımı uygulanmıştır. Bu teknikler, Flutter uygulamalarında gereksiz render, yeniden oluşturma (rebuild) ve yeniden çizim (repaint) işlemlerinin azaltılmasını hedeflemekte olup UI thread üzerindeki yükü doğrudan hafifletmektedir. Aşağıda kullanılan teknikler ayrıntılı biçimde açıklanmıştır.

5.1. const Widget Kullanımı

Flutter'da const ile tanımlanan widget'lar değişmez (immutable) kabul edilir ve uygulama çalıştığı sürece bellekte tek bir örnek olarak saklanır. Bu durum, Flutter'ın widget ağacını oluştururken aynı widget'ın tekrar tekrar yeniden oluşturulmasını engeller.

Bu tekniğin temel katkıları şunlardır:

- Gereksiz *rebuild* işlemlerinin önüne geçilir.
- Özellikle statik içerikli geniş listelerde performans iyileşmesi sağlanır.
- Render süresi azalır, UI thread üzerindeki işlem yoğunluğu düşer.

Bu nedenle optimize edilmiş sürümde sabit widget'lar const ile işaretlenmiş ve performans artışı elde edilmiştir.

5.2. RepaintBoundary Kullanımı

RepaintBoundary, widget ağacını bağımsız çizim alanlarına ayırarak yalnızca değişiklik olan bileşenlerin yeniden çizilmesini sağlar. Bu yaklaşım GPU üzerindeki yükü azaltır ve özellikle çok elemanlı listelerde önemli performans kazancı sağlar.

Bu tekniğin katkıları:

- Tüm liste yerine sadece ilgili satır yeniden çizilir.
- Scroll sırasında ekranın tamamı repaint edilmez.
- GPU yükü azalır, kare süreleri iyileşir.

Bu nedenle optimize sürümde her liste elemanı RepaintBoundary içerisine alınmıştır.

5.3. ListView.builder ile Performanslı Liste Oluşturma

Statik ListView yapısı tüm liste elemanlarını uygulama başında oluşturur. Bu yöntem yüksek bellek kullanımı ve gereksiz render üretir.
Buna karşın ListView.builder yalnızca ekranda görünen elemanları oluşturur; kullanıcı scroll ettikçe diğer elemanlar dinamik olarak oluşturulup yok edilir.

Performansa katkıları:

- Bellek kullanımı azalır.
- İlk açılışta render süresi kısalır.
- Büyük listelerde jank oluşumu engellenir.

Bu nedenle optimize edilmiş uygulamada liste yapısı tamamen ListView.builder üzerine geçirilmiştir.

5.4. Ağır İşlemleri Isolate Üzerine Taşıma (Teorik)

Flutter'da ana thread (UI thread), hem arayüzü oluşturmak hem de animasyonları yönetmekle sorumludur. CPU yoğunluklu hesaplamaların bu thread üzerinde çalıştırılması doğrudan kare sürelerinin artmasına ve jank oluşmasına neden olur.

Bunun çözümü ağır işlemleri isolate yapısı üzerinde çalıştırmaktır. Isolate'lar paralel thread mantığıyla çalışır ve UI thread'inden tamamen ayrıdır.

Bu yaklaşımın katkıları:

- UI thread bloke edilmez.
- Hesaplama süresi uzun olsa bile arayüz akıcı kalır.
- Özellikle gerçek zamanlı scroll veya animasyonlarda stabilite sağlanır.

Bu proje teorik olarak isolate kullanımını içermekte olup, ağır işlem yükünün UI'dan ayrılması gerektiğini göstermektedir.

6. DEĞERLENDİRME VE SONUÇ

Bu proje kapsamında gerçekleştirilen analiz ve uygulama çalışmaları sonucunda, Flutter tabanlı mobil uygulamalarda jank oluşumunun temel olarak üç ana kaynaktan beslendiği görülmüştür: UI thread'in aşırı yüklenmesi, yanlış yapılandırılmış listeleme yöntemleri ve gereksiz rebuild/repaint işlemleri. Özellikle ağır CPU işlemlerinin doğrudan ana thread üzerinde yürütülmesi kare zamanlamalarını olumsuz etkileyerek kullanıcı deneyiminde takılmalara yol açmaktadır. Benzer şekilde, statik ListView kullanımı veya optimize edilmemiş widget ağacı, uygulamanın her karede daha fazla render çalışması yapmasına neden olmaktadır.

Uygulamada gerçekleştirilen optimizasyonlar, söz konusu problemlerin büyük ölçüde azaltabildiğini ortaya koymuştur. Özellikle ListView.builder kullanımıyla yalnızca ekranda görünen elemanların oluşturulması, RepaintBoundary ile yeniden çizim alanlarının sınırlandırılması ve sabit widget'ların const olarak işaretlenmesi sayesinde:

- Kare sürelerinin belirgin biçimde düştüğü,
- Scroll sırasında yaşanan takılmaların büyük ölçüde ortadan kalktığı,
- Uygulamanın genel akıcılığının arttığı,
- Render ve repaint işlem yükünün azaltıldığı

gözlemlenmiştir.

Flutter DevTools araçlarının (Timeline, Repaint Rainbow, Inspector) kullanımı, performans sorunlarının sistematik biçimde tespit edilmesine önemli ölçüde katkı sağlamıştır. Bu araçlar aracılığıyla hem jank'in hangi aşamada oluştuğu hem de hangi widget'ların gereksiz işlem ürettiği ayrıntılı biçimde analiz edilebilmiştir.

Sonuç olarak, bu proje Flutter uygulamalarında performans optimizasyonunun yalnızca kod düzenlemesi ile sınırlı bir süreç olmadığını; doğru araçların, uygun widget kullanımının ve optimizasyon odaklı düşünce yapısının birlikte ele alınması gerektiğini göstermiştir. Uygulanan tekniklerin tamamı, gerçek dünya mobil uygulamalarında karşılaşılan performans sorunlarının giderilmesi açısından pratik ve etkili çözümler sunmaktadır. Bu nedenle, jank azaltma stratejilerinin hem geliştirme sürecinde hem de son test aşamalarında düzenli olarak uygulanması, yüksek kaliteli kullanıcı deneyimi için kritik bir gereklilik olarak değerlendirilmektedir.

KAYNAKÇA

Google Developers. (2024). *Flutter Performance Best Practices*. https://docs.flutter.dev/perf/best-practices

Google Developers. (2024). *Flutter Rendering Pipeline*. https://docs.flutter.dev/resources/rendering

Google Developers. (2024). *ListView & Scrolling Performance*. https://docs.flutter.dev/cookbook/lists/long-lists

Google Developers. (2024). *Performance Profiling with Flutter DevTools*. https://docs.flutter.dev/tools/devtools/performance

Google Developers. (2024). *Widget Rebuild Profiling*. https://docs.flutter.dev/perf/rebuilds

Google Developers. (2024). *Repaint Boundary & Repaint Diagnostics*. https://docs.flutter.dev/tools/devtools/performance#repaints

Flutter Docs. (2023). *Using Isolates for Heavy Computations*. https://docs.flutter.dev/cookbook/networking/background-parsing

Dart Documentation. (2024). *Const Constructors in Dart*. https://dart.dev/language/constructors#const-constructors

Reso Coder. (2021). *When to Use RepaintBoundary in Flutter*. https://resocoder.com/2021/12/10/flutter-repaintboundary-when-to-use/

Flutter Medium Engineering. (2023). *Why Flutter Apps Drop Frames (Jank Explained)*. https://medium.com/flutter/why-flutter-apps-drop-frames-3f8f65b8b3a3

Flutter Community. (2022). *Improving Scroll Performance in Large Lists*. https://medium.com/flutter/improving-scroll-performance-in-flutter-f4b3e3759152